Contents

1 Introduction 6

2 Classes 6
   2.1 Class Structure ........................................ 7
   2.2 Definition of Classes ................................... 8

3 Data Definitions 9
   3.1 Names of Data Items ................................... 10
   3.2 Data Types ............................................. 10
   3.3 Data Declarations ...................................... 11
      3.3.1 Variables of Elementary Types .................... 11
      3.3.2 Object References ................................ 12
   3.4 Scope and Persistence .................................. 12

4 Functions 13
   4.1 Definition of Functions ............................... 13
   4.2 Function Calls .......................................... 14
   4.3 Invoking a Simple Function ............................ 15
   4.4 Data Transfer with Functions ......................... 16
   4.5 Functions with Parameters ............................. 19
      4.5.1 Calling Functions with Arguments ................ 19
      4.5.2 Defining Functions with Parameters .............. 19
   4.6 Initializer Functions .................................. 22

5 Objects 23
   5.1 Creating Objects ..................................... 23
   5.2 Object Interactions ................................. 24

6 Static Methods and Variables 25

7 Assignment 26
   7.1 Arithmetic Expressions .............................. 26
   7.2 Casting ................................................ 27
   7.3 I/O Statements .......................... 27
   7.4 Other Statements with Simple Arithmetic .... 28
   7.5 More Advanced Arithmetic Expressions .......... 29

© 2014 J. M. Garrido
# Selection Statements

- **8.1 Pseudo-code and the IF Statement**
- **8.2 Conditions and Operators**
- **8.3 A Simple Example of Selection**

# If Statement with Multiple Paths

- **9.1 Compound Conditions**
- **9.2 The Case Statement**

# Repetition

- **10.1 Repetition with the While Loop**
- **10.2 Loop Condition and Loop Counter**
- **10.3 Repetition with Loop Until**
- **10.4 Repetition with For Loop**

# Arrays

- **11.1 Declaring Arrays**
- **11.2 Creating Arrays**
- **11.3 Referring Individual Elements of an Array**
  - **11.3.1 Arrays of Simple Types**
  - **11.3.2 Arrays of Object References**
- **11.4 Simple Applications of Arrays**
- **11.5 Finding Maximum and Minimum Values in an Array**
- **11.6 Calculating The Average Value in an Array**
- **11.7 Array Parameters**
- **11.8 Arrays with Multiple Dimensions**

# Strings

- **12.1 Declaring Strings**
- **12.2 String as an Array**
  - **12.2.1 Length of a String**
  - **12.2.2 Retrieving a Character of a String**
- **12.3 Finding the Position of a Character**
- **12.4 Retrieving a Substring from a String**
- **12.5 Finding the Position of a Substring**
- **12.6 Joining Two or More Strings**
- **12.7 Comparing Strings**
# OOSimL Technical Report

## 13 Inheritance
- 13.1 Classification ........................................ 57
- 13.2 Base Class ............................................. 57
- 13.3 Defining new Classes with Inheritance ................. 58
- 13.4 Initializer Functions in Subclasses .................... 59

## 14 Abstract Classes
- 14.1 Abstract Classes ...................................... 60
- 14.2 Defining an Abstract Class ............................ 61

## 15 Interfaces
- 15.1 Using an Interface .................................... 63
- 15.2 Subtypes ............................................... 64

## 16 Polymorphism
- 16.1 Simple Application of Polymorphism .................. 65
- 16.2 Heterogeneous Array .................................. 66

## 17 Basic Graphical User Interfaces
- 17.1 Graphical Objects ..................................... 67
- 17.2 Components and Containers ............................ 68
- 17.3 Using Graphic Libraries ............................... 68
- 17.4 Frames ................................................. 69
- 17.5 Simple Components ................................... 70
- 17.6 Adding Components to a Window ...................... 71
- 17.7 Attributes of Frames .................................. 74
- 17.8 Events and Listeners .................................. 75
- 17.9 Objects that Generate Events ......................... 75
- 17.10 Adding a Button to a Window ....................... 76
- 17.11 Data Input ............................................ 79
- 17.12 Decimal Formatting ................................... 80
- 17.13 Applets ............................................... 81
- 17.14 Panel Containers ..................................... 83
- 17.15 Drawing Simple Objects .............................. 86
- 17.16 General Functions for Drawing ....................... 87

## 18 Exceptions and I/O
- 18.1 Dealing with Exceptions .............................. 88
- 18.2 Checked and Unchecked Exceptions ................... 89
- 18.3 Basic Handling of Exceptions ......................... 89

© 2014 J. M. Garrido
18.4 Files ........................................ 91
18.5 Text and Binary Files ............................ 91
18.6 Handling Text Files ............................... 92
18.7 Output Text Files ................................. 92
18.8 Input Text Files ................................ 93
18.9 Files with Several Values on a Text Line .... 95
1 Introduction

The Object-Oriented discrete-event Simulation Language, OOSimL, was developed as an educational tool and to help system modelers design and implement simulation models using the object-oriented approach to software development. OOSimL supports the process interaction approach to discrete-event simulation. This allows higher-level simulation modeling and the flexibility of including as much detail as needed.

Programs written with OOSimL can be easily integrated with Java programs. The OOSimL compiler generates Java code. The run-time support of the language is a major enhancement of the PsimJ simulation package, which is a collection of Java classes.

This document is Part I of the reference manual, which presents the object-oriented programming aspects of OOSimL. For the simulation aspects of OOSimL, refer to Part II of this reference manual.

The OOSimL compiler, run-time libraries, and related documentation are freely available for educational and research purposes only. The software and documentation are copyrighted (©J Garrido) materials and are part of the OOPsim project. The most recent versions of these materials can be downloaded with examples from the OOPsim Web page; the URL for the Web pages is as follows:

ksuweb.kennesaw.edu/~jgarrido/psim.html

Programming languages, including OOSimL, have well-defined syntax and semantic rules. The syntax is defined by a set of grammar rules and a vocabulary (a set of words). The legal sentences are constructed using words in the form of statements. The set of words is divided into two groups:

1. Reserved words (or keywords) have a predefined purpose in the language and are used in most statements with precise meaning, for example, class, inherits, variables, while, if, and so on.

2. Identifiers are names for variables, constants, functions, and classes that the programmer chooses, for example, lserver, car, server_name, and Server.

2 Classes

Classes are reusable units, which means that they can be used in another application. Classes are static definitions and are considered the main decomposition units; every program is composed of one or more classes. A
class is a compilation unit; the compilation of a program is carried out by compiling every class of the program. A class is also considered a type for objects.

2.1 Class Structure

A class defines the attributes and behavior of the objects in a collection. In other words, every collection of entities is defined by describing the attributes and behavior of these entities. The attributes are defined as data declarations, and the behavior is defined as one or more operations (also known as methods and functions). Figure 1 shows the general structure of a class.

![Figure 1: General structure of a class named Class_A.](image)

A software object is described as an encapsulation of the attributes and behavior of the object into a single unit. When access to some of the attributes and some operations is not allowed, the access is said to be private; otherwise, the access mode is public. This is considered an encapsulation protection mechanism.

Applying the principle of information hiding, knowledge about an object is limited; only the knowledge necessary for the features of an object to be

© 2014 J. M. Garrido
accessed by another object is made available. The rest of the knowledge about the object is not revealed. The internal details of how the various operations are carried out are not made available to other objects.

If a function of an object is *public*, it is accessible from another object; the implementation details are kept hidden. In general, only the function headers (also known as function specifications) are known to other objects.

The software definition of a class consists of:

- The data declaration of the attributes of the class
- A set of method or function definitions

The attributes in a class are defined as variable declarations. The data definitions inside a function are written as local variable declarations. Object-oriented programs include the following kinds of statements:

- Class definitions
- Declaration of simple variables
- Declaration of object references
- Definition of functions
- Creation of objects
- Manipulation of the objects created by calling (or invoking) the functions that belong to these objects

In OOSimL, the functions or operations are known as methods. Objects do not directly have names, instead, object reference variables are used to reference objects.

### 2.2 Definition of Classes

A class definition in OOSimL includes several sections. These must be written in the following order:

1. The optional *description* statement encloses a textual description of the purpose of the class, author, date, and any other relevant information. This section ends with a star-slash (/).  

2. The *class* statement defines the name of the class and other information related to the class.
3. The **private** section includes data declarations of the private attributes of the class and definitions of the private functions or methods of the class.

4. The **public** section includes data definitions of public attributes and the public definitions of functions or methods of the class.

5. The **endclass** statement ends the class definition.

The general syntactic definition of a class is:

```
description

class ( class_name ) is
    private
        constants
        variables
        object references
    [function definitions]
    public
    [data and function definitions]
endclass ( class_name )
```

Functions are not reusable units because every function belongs to a class. From this point of view, a function is an internal decomposition unit.

### 3 Data Definitions

Data consists of one or more data items. For every data item, its definition is given by:

- A unique name to identify the data item
- An optional initial value
- The data item type
The name of a data item is an identifier and is given by the programmer; it must be different from any keyword in OOSimL. The type defines:

- The set of possible values that the data item may have
- The set of possible operations that can be applied to the data item

### 3.1 Names of Data Items

The special symbols that indicate essential parts of an algorithm are called *keywords*. These are reserved words and cannot be used for any other purpose. The other symbols used in an algorithm are the ones for identifying the data items and are called *identifiers*. The identifiers are defined by the programmer.

A unique name or label is assigned to every data item; this name is an identifier. The problem for calculating the area of a triangle used five data items, \(x\), \(y\), \(z\), \(s\), and \(area\).

Variables are data items that change their values when they are manipulated by the various operations. For example, the following sequence of instructions first gets the value of \(x\) then adds the value \(x\) to \(y\):

```plaintext
read time // read value of time from keyboard
add time to acctime
```

Those data items that do not change their values are called *constants*, for example, \(Max\_period\), \(PI\), and so on. These data items are given an initial value that will never change.

### 3.2 Data Types

There are two broad groups of data types:

- Elementary (or primitive) data types
- Classes

Elementary types are classified into the three following categories:

- Numeric
- Text
Booleans

The numeric types are further divided into three types, integer, float, and double. The noninteger types are also known as fractional, which means that the numerical values have a fractional part.

Values of integer type are those that are countable to a finite value, for example, age, number of automobiles, number of pages in a book, and so on. Values of type float have a decimal point; for example, cost of an item, the height of a building, current temperature in a room, a time interval (period). These values cannot be expressed as integers. Values of type double provide more precision than type float, for example, the value of the total assets of a corporation.

Text data items are of two basic types: character and type string. Data items of type string consist of a sequence of characters. The values for these two types of data items are textual values.

A third type of variables is the one in which the values of the variables can take a truth-value (true or false); these variables are of type boolean.

Classes are more complex types that appear as types of object reference variables in all object-oriented programs. Data entities declared (and created) with classes are called objects.

3.3 Data Declarations

The data definitions are the data declarations. Each data definition includes the name of every variable or constant with its type. The initial values, if any, for the data items are also included in the data declaration. There are two general categories of variables:

- Elementary
- Object variables (references)

Object-oriented programming is mainly about defining classes as types for object variables (references), and then declaring and creating objects of these classes. The type of an object reference is a class.

3.3.1 Variables of Elementary Types

The OOSimL statement for the declaration of variables of elementary types has the following structure:

\[
\text{define } \langle \text{variable name} \rangle \text{ of type } \langle \text{elementary type} \rangle
\]
The following are examples of data declarations in OOSimL of two constants of types integer and double, followed by three elementary variables of type integer, float, and boolean.

```plaintext
constants
  define MAX_PERIOD = 24 of type integer
  define PI = 3.1416 of type double
variables
  define count of type integer
  define salary of type float
  define act_flag of type boolean
```

### 3.3.2 Object References

As mentioned previously, an object reference is a variable that can refer to (point to) an object. The OOSimL statement for declaration of object references has the following structure:

```plaintext
define ⟨ object_ref_name ⟩ of class ⟨ class_name ⟩
```

For example, consider a program that includes two class definitions, Server and Car. The declarations of an object reference called server_obj of class Server, and an object reference car1 of class Car are:

```plaintext
object references
  define server_obj of class Server
  define car1 of class Car
```

### 3.4 Scope and Persistence

When identifying data items in software development, there are two important concepts to consider:

- The **scope** of a data item is that portion of a program in which statements can reference that data item
• The persistence of a data item is the interval of time that the data item exists—the lifetime of the data item

4 Functions

A program is normally decomposed into classes, and the classes consist of data declarations and definitions of functions (or methods). Functions or methods are the internal decomposition units. A function represents a small task.

4.1 Definition of Functions

Figure 2 illustrates the general structure of a function. The body of a function consists of two basic parts:

• The data declarations

• The instructions that manipulate the data

<table>
<thead>
<tr>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data declarations</td>
</tr>
<tr>
<td>Instructions</td>
</tr>
</tbody>
</table>

Figure 2: General structure of a function.

The data declared within a function is known only to that function—the scope of the data is local to the function. Every function has its own data declaration and instructions that manipulate the data. The data in a function only exists during execution of the function; their persistence is limited to the lifetime of the function.

The purpose of the function is described in the optional paragraph that starts with the keyword description and ends with a star-slash (/). The
name of the function is declared in the function statement. This name is used when the function is called or invoked by some other function. The following OOSimL code defines the basic structure of a function.

A function can receive input data from another function when invoked; the input data passed from another function is called a parameter. The function can also return output data when it completes execution.

\[
\text{description} \\
\text{function (function_name) is} \\
\text{constants} \\
\text{variables} \\
\text{object references} \\
\text{begin} \\
\text{endfun (function_name)}
\]

The header of a function starts with the keyword function followed by the name of the function; the body of the function starts with the keyword is. The function ends with the endfun keyword and the name of the function. The keywords method and endmethod are also valid.

The data declarations are divided into constant, variable, and object declarations. This is similar to the data declarations in the class. Constant, variable, and object declarations are optional. The instructions in the body of the function appear between the keywords begin and endfun. Line comments start with a double forward slash (//) and end with the line.

One of the classes in every program must include a function called main. This function starts and terminates the execution of the entire program; it is the control function of the program.

In very simple and small programs that have only one class, the algorithm for the problem is implemented in function main—all the instructions for the solution of the problem are located in this function.

4.2 Function Calls

In every function call (or method invocation), the function that calls another function is known as the calling function; the second function is known as the called function. When a function calls or invokes another function, the normal execution flow of control in the first function is interrupted. The flow of control is altered and the second function starts execution. When
the called function completes execution, the flow of control is transferred back (returned) to the calling function. This function continues execution from the point after it called the second function.

The most obvious categories of functions discussed so far are the private and public functions. Only the public functions of an object can be invoked from another object. The private functions are sometimes called internal functions because they can only be invoked by another function of the same object.

The instructions in a function start execution when the function is called or invoked. After completion, the called function may or may not return a value to the calling function. From the data transfer point of view, there are three general categories of functions:

1. Simple (or void) functions do not return any value when they are invoked.

2. Value-returning functions return a single value after completion.

3. Functions with parameters require one or more data items as input values when invoked.

The most general category of functions is one that combines the last two categories just listed—functions that return a value and that have parameters.

Another important criterion for describing categories of functions depends on the purpose of the function. There are two such categories of functions:

- Accessor functions return the value of an attribute of the object without changing the value of any attribute(s) of the object.

- Mutator functions change the state of the object in some way by altering the value of one or more attributes in the object. Normally, these functions do not return any value.

It is good programming practice to define the functions in a class as being either accessor or mutator.

4.3 Invoking a Simple Function

A simple (also called void) function does not return a value to the calling function. The simplest function of this kind is function \textit{display} _\textit{message},
discussed previously. There is no data transfer to or from the function. The OOSimL statement to call or invoke a void function that is referenced by an object reference is:

\[
\text{call } \langle \text{function\_name} \rangle \text{ of } \langle \text{object\_ref} \rangle
\]

For example, suppose the function \textit{display\_message} that belongs to an object referenced by \textit{myobj} is invoked from function \textit{main}, the call statement is:

\[
\text{call display\_message of myobj}
\]

Figure 3 shows the calling mechanism. In the figure, the calling function is \textit{main}, and the called function is \textit{display\_message}. After completing its task, the called function returns the flow of control to the calling function.

![Figure 3: Calling a function.](image)

### 4.4 Data Transfer with Functions

A more complex mechanism in calling a function involves data transfer between the calling and the called function. This data transfer is possible in two directions, from the calling function to the called function and from the called function (return value) to the calling function. The simplest data transfer involves value-returning functions in which the direction of the data transfer is from the called function to the calling function.

With value-returning functions, some value is calculated or assigned to a variable that is returned to the calling function. Note that the return value
is a single value; no more than one value can be returned.

A type defined as \( \text{function\_type} \) in the called function, can be a simple type or a class and is the type of the return value. This return value is actually the result produced by the called function. After the called function completes, control is returned with a single value to the calling function. The general structure of a value-returning function in OOSimL is:

\[
\text{description} \\
\text{*} \\
\text{function} \langle \text{funct\_name} \rangle \text{ return type } \langle \text{function\_type} \rangle \text{ is} \\
\text{ . . . .} \\
\text{return} \langle \text{return\_value} \rangle \\
\text{endfun} \langle \text{function\_name} \rangle
\]

The value in the return statement can be any valid expression, following the \text{return} keyword. The expression can include constants, variables, object references, or a combination of these.

For example, suppose a function with name \text{get\_val} displays a message on the console asking for an \text{integer} value; this value is read from the console and is returned to the calling function. In the header of the function, the type of the value returned is indicated as \text{integer}. The OOSimL code for this function definition is:

\[
\text{description} \\
\text{This function asks the user for an integer value, then this value is returned. *} \\
\text{function get\_val return type integer is} \\
\text{variables} \\
\text{ define local\_var of type integer} \\
\text{begin} \\
\text{ display "Please enter an integer value: "} \\
\text{ read local\_var } // \text{ read value from console} \\
\text{ return local\_var } // \text{ return value read} \\
\text{endfun get\_val}
\]

The calling function can call a value-returning function, and the value returned can be used in one of the following ways:

- A simple assignment statement

© 2014 J. M. Garrido
• An assignment with an arithmetic expression

In a simple assignment, the value returned by the called function is used by the calling function by assigning this returned value to another variable.

\[
\text{call } \langle \text{fun_name} \rangle [\text{of } \langle \text{object_ref} \rangle][\text{value}][\text{to } \langle \text{var_name} \rangle]
\]

For example, suppose function main calls function get_val of an object referenced by myobj. Function main assigns the value returned to variable y. There several ways in which OOSimL can be used to implement this call. The OOSimL code statements for these calls are:

\[
call \text{get_val of myobj value to y}
\]
\[
\text{set } y = \text{call get_val of myobj}
\]
\[
\text{set } y = \text{myobj.get_val()}
\]

This call statement is included in an assignment statement in function main. When the call is executed, the sequence of activities that are carried out is:

1. Function main calls function get_val in object referenced by myobj.
2. Function get_val executes when called and returns the value of variable local_var.
3. Function main receives the value returned from function get_val.
4. Function main assigns this value to variable y.

Using the value returned in an assignment with an arithmetic expression is straightforward after calling the function. For example, after calling function get_val of the object referenced by myobj, the value returned is assigned to variable y. This variable is then used in an arithmetic expression that multiplies y by variable x and adds 3. The value that results from evaluating this expression is assigned to variable zz. This assignment statement is:

\[
\text{set } zz = x \ast y + 3
\]
4.5 Functions with Parameters

Most useful functions can receive data values when called from another function. These data values are treated as input values by the called function. The data definitions for these input values are called parameter declarations and are similar to local data declarations. Local data has a local scope and the persistence is for the duration of the function execution. If the called function returns a value, it can be of any legal type but not of type \texttt{void}.

4.5.1 Calling Functions with Arguments

The called function can be supplied with the data values. These values are known as \textit{arguments} and can be actual values (constants) or names of data items. The function call includes the keyword \texttt{using} and this is followed by the list of arguments, which are values or names of the data items. When there are two or more argument values in the function call, the argument list consists of the data items separated by commas. The OOSimL statement for a function call with arguments is:

\begin{verbatim}
call ⟨function_name⟩ of ⟨object_ref⟩
    using ⟨argument_list⟩
\end{verbatim}

For example, consider a call to function \texttt{min.1} that belongs to an object referenced by \texttt{obj.a}. The function calculates and prints the minimum value of the two given arguments \texttt{x} and \texttt{y}. This call statement in OOSimL is:

\begin{verbatim}
call min.1 of obj.a using x, y
\end{verbatim}

4.5.2 Defining Functions with Parameters

The definition of a function with parameters includes the declaration of the data items defined as parameters. For every parameter, the function declares a name and its type. The parameter list follows the keyword \texttt{parameters}. The general structure of a function with parameter definition is:

\begin{verbatim}
description
...
*/

function ⟨function_name⟩
    parameters ⟨parameter_list⟩ is
    ...
\end{verbatim}
The following example uses this syntax structure with a function named \textit{min$_1$}. The header for the definition of function \textit{min$_1$} in OOSimL is:

\begin{verbatim}
function min_1 parameters a of type real, b of type integer is
\end{verbatim}

Figure 4: Transferring arguments in a function call.

The definition of function \textit{min$_1$} declares the parameters \textit{a} and \textit{b}. These parameters are used as placeholders for the corresponding argument values transferred from the calling function. Figure 4 illustrates the calling of function \textit{min$_1$} with the transfer of argument values from the calling function to the called function.

In the call to function \textit{min$_1$} shown in Figure 4, the value of argument \textit{x} is assigned to parameter \textit{a}, and the value of argument \textit{y} is assigned to parameter \textit{b}. The arguments and the parameters must correspond in type and meaning, so the order of arguments in the argument list depends on the parameter list definition. The general structure of a value-returning function with parameter definition is:

\begin{verbatim}
endfun ( function_name )
\end{verbatim}
For example, consider a function called \texttt{min\_2} that returns the minimum of two integer values. The OOSimL definition for this function is:

\begin{verbatim}
description
   This function calculates the minimum value of parameters \texttt{x} and \texttt{y}, it then returns this value.

function min_2 return type integer parameters
   x of type integer, y of type integer is
variables
   define min of type integer // local variable
begin
   if \texttt{a} < \texttt{b} then
      set \texttt{min} = \texttt{x}
   else
      set \texttt{min} = \texttt{y}
   endif
   return \texttt{min}
endfun min_2
\end{verbatim}

The call to function \texttt{min\_2} appears in an assignment statement. For example, to call \texttt{min\_2} with two constant arguments, \texttt{a} and \texttt{b}, and assign the return value to variable \texttt{y}:

\begin{verbatim}
set y = call min_2 of obj\_a using \texttt{a}, \texttt{b}
\end{verbatim}

The following portion of code shows another way to use the \texttt{call} statement and get the same result.

\begin{verbatim}
call min_2 of obj\_a using \texttt{a}, \texttt{b} value to \texttt{y}
\end{verbatim}
4.6 Initializer Functions

A class definition normally includes one or more initializer functions, also known as constructors. These constitute a special group of functions that are called when creating objects of the enclosing class. The main purpose of an initializer function is to set the object created to an appropriate (initial) state. It carries this out by assigning initial values to the attributes of the object.

The following example includes the partial definition of a class named Person that two private attributes, age and obj_name.

class Person is
private
  variables // variable data declarations
    define age of type integer
    define obj_name of type string
public
  function initializer parameters iage of type integer
    iname of type string is
begin
  set age = iage
  set obj_name = iname
endfun initializer

endclass Person

An initializer function is included in class Person to set given initial values to the attributes of the object when created. If no initializer function is defined in the class, the default values for the attributes are set by the compiler when creating an object of the class. The general structure of the statement to create an object is:

    create ( object_ref_name ) of class ( class_name )

This statement implicitly calls the default initializer function in the class. An explicit initializer function sets the value of all the attributes to the values given when called. These values are given as arguments in the statement that creates the object. The general statement to create an object with given values for the attributes is:

    create ( object_ref_name ) of class ( class_name )
using \{ argument_list \}

Suppose there is an object reference `person_obj` of class `Person` declared, to create an object referenced by `person_obj` with initial value of 32 for `age` and "J. K. Hunt" for the `obj_name`:

```oosiml
create person_obj of class Person using 32, "J. K. Hunt"
```

The definition of two or more functions with the same name is known as overloading. This facility of the programming language allows any function in a class to be overloaded. In other words, it is a redefinition of a function with another function with the same name, but with a different number of and types of parameters.

It is very useful in a class to define more than one initializer function. This way, there is more than one way to initialize an object of the class. When a class defines two initializer functions, the compiler differentiates them by the number of and the type of parameters.

## 5 Objects

### 5.1 Creating Objects

After declaring object references, the corresponding objects can be created in the body of a function. The OOSimL statement to create objects has the following structure:

```oosiml
create \{ object_ref_name \} of class \{ class_name \}
```

For example, assume that the program described previously is to create an object for each of the two class definitions, `Employee` and `Ball`. The declarations of the object reference called `emp_obj` of class `Employee`, and the object reference `ball1` of class `Ball` were previously provided. The OOSimL statements for creating the two objects are:

```oosiml
create server_obj of class Server
create car1 of class Car
```

Function `main` is a special function; execution of the entire application starts in this function, and terminates here. This function must have been defined as `public`; otherwise, the function is not accessible to other objects. In a typical program (or application), function `main` carries out the following general sequence of tasks:

© 2014 J. M. Garrido
1. Declares constants, simple variables, and object variables, as necessary

2. Creates one or more objects of the previously defined classes

3. Invokes one or more methods of the various objects. This delegates the tasks and subtasks to the objects for carrying the complete solution of the application.

5.2 Object Interactions

Objects interact with one another by sending and receiving messages to and from each other. The object that sends the message is the requestor (or the client) of a service, which can be provided by the receiver object (the provider). A message represents a request for service, which is provided by the object receiving the message. The sender object is known as the client of a service and the receiver object as the supplier of the service.

The purpose of sending a message is to request some operation of the receiver object to be carried out; in other words, the request is a call to one of the operations of the supplier object. This is also known as method invocation. Objects execute operations in response to messages and the operations carried out are object specific.

For example, an object of class Userproc sends a message to an object of class Server. In this message, some function (operation) of the first object calls function of the second object. A message is always sent to a specific object, and it contains four parts:

- The function (operation) to be invoked or started, which is the service requested; this must be a public function if invoked by another object
- The input data required in order for the operation to start; this data is known as the arguments
- The output data, which is the reply to the message; this is the actual result of the request
- The optional reference to the receiver (supplier) object.

To describe the general interaction between two or more objects (the sending of messages between objects), the UML collaboration diagram is used; see Figure 5 as an example. Another UML diagram used to describe object interaction is the sequence diagram.
6 Static Methods and Variables

If a method does not belong to an object, but it is defined in its class, the method is normally called with the class name, a dot, and then the method name. No object needs to be referenced when calling such a method. For example:

\[
\text{set } y = \text{Math.sin}(x) + 109.5
\]

In this case, \text{sin} is a static method of the Java class \text{Math}. To define a static method, the keyword \text{static} is written before the name of the method. For example:

\begin{verbatim}
function static mymethod is
  ...
endfun mymethod
\end{verbatim}

Static variables are not associated with any object. These variables should normally be private and should be accessed and/or updated using accessor and mutator functions. To declare a static variable in a class, the keyword \text{static} is written before the normal variable declaration. For example, the following statement declares a static variable named \text{num_var} and initializes it to a value of zero.

\[
\text{define static num_var = 0 of type integer}
\]

For every declaration of a static variable, there is only one copy of its value shared by all objects of the class.
A static method cannot reference nonstatic variables and cannot include calls to nonstatic methods. The basic reason for this is that nonstatic variables and methods belong to an object, static variables and methods do not.

Often, static variables are also known as class variables, and static methods as class methods.

7 Assignment

An assignment statement is used to give a value to a variable, and this means that during execution of a program, the value of the variable changes. The keyword `set` must be included at the beginning of the assignment statement. There are two basic ways to give a new value to a variable:

1. Simple assignment

2. The result of evaluating an expression is assigned to a variable

7.1 Arithmetic Expressions

The simple assignment simply gives or sets a constant value to a variable. For example, given the following declarations:

```
variables
   define length of type real
   define x  of type integer
```

The assignment statement to assign the constant value 45.85 to variable `length` is:

```
set length = 45.85
```

In a similar manner, the assignment statement to give the value 54 to variable `x` is:

```
set x = 54
```

In writing an assignment statement, the variable that is receiving the new value is always placed in the left-hand side of the equal sign, the assignment operator.
The second kind of assignment statement is a more general assignment and is used when the value assigned to a variable is the result of evaluating an expression. For example, the value that results from evaluating the expression \( x + 4.95z \) is assigned to variable \( y \). The assignment statement is:

\[
\text{set } y = x + 4.95 \times z
\]

### 7.2 Casting

Casting is sometimes necessary in an assignment statement to convert the default type of an expression to a compatible type for the variable to receive the value. For example, the type conversion of an expression from `double` to `integer`. The same is applied for class conversion.

\[
\text{set } \langle \text{variable name} \rangle = \text{cast type } \langle \text{data type} \rangle \langle \text{expression} \rangle \\
\text{set } \langle \text{variable name} \rangle = \text{cast class } \langle \text{class name} \rangle \langle \text{expression} \rangle
\]

The following lines of code declare two variables of different types and the casting is included in an assignment using the `set` statement.

```
define j of type integer
define area of type double

set j = cast type integer area \times 12.5
```

### 7.3 I/O Statements

For console applications, the two simple statements for input/output require the keywords `read`, for input and `print` for output. The input statement allows the algorithm to read a value of a variable from the input device (e.g., the keyboard). The value is assigned to the variable indicated. The general structure of the input statement is:

```
read \langle \text{variable name} \rangle
```

For example, to read a value for variable `length`, the statement is:

```
read length
```

This is similar to an assignment statement for variable `length`, since the variable changes its value to the new value that is read from the input device.
The output statement writes the value of a variable to the output device (e.g., the video screen). The variable does not change its value. The general structure of the output statement is:

\[ \text{display } \langle \text{data_list} \rangle \]

The data list consists of the list of data items separated with commas. For example, to print the value of variable \(X\) on the video screen unit, the statement is:

\[ \text{display } X \]

A more practical output statement that includes a string literal and the value of variable \(X\) is:

\[ \text{display } "\text{value of } X \text{ is: }", X \]

### 7.4 Other Statements with Simple Arithmetic

The following statements are similar to the ones previously discussed and they involve only simple arithmetic operations.

- **add** 24 to \(x\)
- **increment** \(j\)
- **subtract** \(x\) from \(y\)
- **decrement** \(\text{counter}_a\)

The first statement takes the constant value 24 and adds it to the current value of variable \(x\). The result of the addition becomes the new value of variable \(x\). This statement requires the keyword **add** and is equivalent to the following assignment statement:

\[ \text{set } x = x + 24 \]

On the right-hand side of the equal sign (the assignment operator), the current value of variable \(x\) is used. The variable on the left-hand side of the assignment operator changes its value; variable \(x\) now has a new value.

The **increment** statement, **increment** \(j\), adds the constant 1 to the current value of variable \(j\). This statement is equivalent to the following assignment statement:
set \ j = j + 1

The other two statements, \texttt{subtract} and \texttt{decrement}, are applied in a similar manner to the first two.

### 7.5 More Advanced Arithmetic Expressions

The arithmetic expressions used in the assignment statements discussed previously were very simple; only basic arithmetic operations appeared in the expressions. These are addition, subtraction, multiplication, and division.

To use a particular function of this library class, the name of this class must appear followed by a dot then followed by the name of the particular function invoked. For example, consider the value of the expression $\sqrt{s}$ that is assigned to variable $y$. The expression in the assignment statement must use the mathematical function $\texttt{sqrt}$ in class \texttt{Math}; the statement is:

```plaintext
set y = Math.sqrt(s)
```

Class \texttt{Math} is a pre-defined class and is part of the class library supplied with the Java compiler. This class provides several mathematical functions such as square root, exponentiation, trigonometric and other mathematical functions.

In a similar manner, to assign to variable $\text{interest}$ the value of the mathematical expression $x^2$, the complete assignment statement is:

```plaintext
set interest = Math.pow(x, 2)
```

This statement invokes function $\texttt{pow}$ from class \texttt{Math}, to raise the value of $x$ to the power of two; the value of this expression is assigned to variable $\text{interest}$.

### 8 Selection Statements

The selection statements include conditions, which are Boolean expressions that evaluate to a truth-value (true or false). Simple conditions are formed with relational operators for comparing two data items. Compound conditions are formed by joining two or more simple conditions with the \textit{logical} operators.

The selection design structure is also called alternation because alternate paths are considered based on a condition. This structure is easier to understand in a flowchart. Figure 6 shows two possible paths for the
execution flow. The condition is examined (or evaluated) and a decision is made to select one of the paths. If the condition is true, then the left path is taken and the instruction on this path (Block1) will be executed. If the condition is not true, the other path is taken and the instructions on this path (Block2) will be executed. Thus, the selection structure provides the algorithm capability for decision-making.

Figure 6: Flowchart segment general selection structure.

8.1 Pseudo-code and the IF Statement

In the pseudo-code notation, the selection structure is written with an if statement, also called an if-then-else statement. This statement includes several keywords; recall that these are reserved words because the programmer cannot use any of these words for other purposes. The keywords are: if, then, else, and endif.

In OOSimL, the general structure of the if statement is:

```plaintext
if ( condition )
    then
        ⟨ statements in Block1 ⟩
    else
        ⟨ statements in Block2 ⟩
```

© 2014 J. M. Garrido
The informal pseudo-code for the if statement that corresponds to the general selection structure illustrated in Figure 6 is:

```c
if condition is true
    perform instructions in Block1
else
    perform instructions in Block2
endif
```

The if statement is considered a compound statement. All the instructions in Block1 are said to be in the then-section of the if statement. In a similar manner, all the instructions in Block2 are said to be in the else-section of the if statement.

When the if statements executes, the condition is evaluated and only one of the two alternatives will be carried out; the one with the statements in Block1 or the one with the statements in Block2.

### 8.2 Conditions and Operators

The condition consists of an expression that evaluates to a truth-value, **true** or **false**. These types of expressions are also known as Boolean expressions. A simple Boolean expression compares the value of two data items.

A simple Boolean expression is composed of two data items and a relational operator to compare the two data items. There are six relational operators, and these are:

- Equal, `==`
- Not equal, `!=`
- Less than, `<`
- Less or equal to, `<=`
- Greater than, `>`
- Greater or equal to, `>=`

Examples of simple conditions that can be expressed with the relational operators are:
Instead of applying the mathematical symbols shown above for the relational operators, additional keywords can be used for the operators in OOSimL. For example:

\[ X \text{ \textgreater or equal to } Y \]
\[ \text{time } \text{not equal } \text{start}_t \]
\[ a \text{ \textgreater than } b \]

### 8.3 A Simple Example of Selection

Consider a portion of an algorithm in which the decision whether variable \( j \) should be incremented or decremented depends on the condition: \( x > 0 \). Figure 7 illustrates the flowchart portion of the algorithm that includes this selection structure for this simple example.

For the example described above, the portion of the algorithm written in pseudo-code is:
if $x > 0$
then
increment $j$
else
decrement $j$
endif

In the selection statement, only one of the two paths will be taken; this means that in the example, the statement increment $j$ or the statement decrement $j$ will be executed. This depends on the evaluation of the condition $x > 0$.

9 If Statement with Multiple Paths

The if statement with multiple paths is used to implement decisions involving more than two alternatives. The additional elseif clause is used to expand the number of alternatives. The general syntax of the if statement with $k$ alternatives is:

\[
\text{if } (\text{condition}) \\
\text{then} \\
\quad (\text{sequence1 of statements}) \\
\text{elseif } (\text{condition2}) \\
\text{then} \\
\quad (\text{sequence2 of statements}) \\
\text{elseif } (\text{condition3}) \\
\text{then} \\
\quad (\text{sequence3 of statements}) \\
\quad ... \\
\text{else} \\
\quad (\text{sequencek of statements}) \\
\text{endif}
\]

The conditions in a multiple-path if statement are evaluated from top to bottom until one of the conditions evaluates to true. The following example applies the if statement with four paths.

if height > 6.30
then increment group1
elseif height > 6.15
then increment group2
elseif height > 5.85

© 2014 J. M. Garrido
then increment group3
elseif height > 5.60
then increment group4
else increment group5
endif

9.1 Compound Conditions

More complex expressions can be constructed with the logical operators and, or, and not. These logical operators help to join two or more simple conditions to construct conditions that are more complex.

The general structure of a compound condition using the or operator and two simple conditions, cond1 and cond2, is:

\[ \text{cond1 or cond2} \]

The other two logical operators are used in the same manner. The following lines of code show an example of a compound condition.
if \( a \neq b \) and \( x > 0 \) then
\( \langle \text{statements}_1 \rangle \)
else
\( \langle \text{statements}_2 \rangle \)
endif

This same compound condition can be constructed in a more verbose manner. For example:

if \( a \) not equal \( b \) and \( x \) greater than \( 0 \)...

The following expression uses the \textbf{not} operator:

\[
\textbf{not} \ (x \leq y)
\]

This expression has the following descriptive meaning: it is not true that \( x \) is less than or equal to \( y \).

\section*{9.2 The Case Statement}

The case structure is an expanded selection structure because it has more than two alternate paths. The case statement tests the value of a variable of type \textit{integer} or of type \textit{character}. Depending on the value of this variable, the statement selects the appropriate path to follow.

The variable to test is called the selector variable. The general structure of the \texttt{case} statement is:

\[
\texttt{case} \ \langle \texttt{selector\_variable} \rangle \ \texttt{of} \\
\texttt{value} \ \texttt{sel\_variable\_value} : \ \langle \texttt{statements} \rangle \ldots
\]

\texttt{endcase}

The following example determines the numeric value of the letter grades. Assume the possible letter grades are A, B, C, D, and F, and the corresponding numerical grades are 4, 3, 2, 1, and 0. The following case statement first evaluates the letter grade in variable \texttt{letter\_grade} and then, depending on the value of \texttt{letter\_grade}, it assigns a numerical value to variable \texttt{num\_grade}.

\[
\texttt{case \ letter\_grade \ of}
\texttt{value} \ 'A' : \ \texttt{num\_grade} = 4 \\
\texttt{value} \ 'B' : \ \texttt{num\_grade} = 3
\]
10 Repetition

Most practical algorithms require a group of operations to be carried out several times. The repetition design structure is a looping structure in which a specified condition determines the number of times the group of operations will be carried out. This group of operations is called a repetition group.

There are three variations of the repetition structure and most programming languages include them. This section explains three constructs to apply the three variations of the repetition structure. The constructs for repetition are:

1. While loop
2. Loop until
3. For loop

The first construct, the while-loop, is the most general one. The other two repetition constructs can be expressed with the while-loop.

10.1 Repetition with the While Loop

The repeat structure requires a loop condition and a repeat group. With the while-loop construct, the loop condition is tested first. If the condition is true, the operations in the repeat group are carried out. This continues until the condition evaluates to false.

Figure 8 shows a flowchart segment that illustrates the while-loop construct. The repeat group consists of the operations in Block1.

If the condition is true, the operations in Block1 are executed, then the condition is again evaluated and the operations are carried out if the condition is still true. This continues until the condition changes to false, the loop terminates, and then the control flow continues with the operation that follows the while loop construct.

In OOSimL, the while statement is written with the keywords while, do, and endwhile. The repeat group consists of all operations in Block1,
which is placed after the `do` keyword and before the `endwhile` keyword. The following portion of code shows the general structure for the while-loop construct with the while statement in OOSimL that corresponds to the portion of flowchart shown in Figure 8.

```plaintext
while ( condition ) do
    ( statements in Block1 )
endwhile
```

### 10.2 Loop Condition and Loop Counter

Note that in the while-loop construct the condition is tested first, then the repeat group is carried out. If this condition is initially false, the operations in the repeat group will not be carried out.

The number of times that the loop is carried out will normally be a finite integer value. This implies that the condition will eventually be evaluated to false, i.e., the loop will eventually terminate. This condition is sometimes called the *loop condition*, and it determines when the loop terminates. Only in some very special cases the programmer can decide to write an infinite loop, this will repeat the operations in the repeat loop forever.
A counter variable has the purpose of storing the number of times that some condition occurs in a function. The counter variable is of type integer and is incremented every time some specific condition occurs. The variable must be initialized to a given value.

A loop counter is an integer variable that is incremented every time the operations in the repeat group are carried out. Before starting the loop, this counter variable must be initialized to some particular value.

The following portion of code has a while statement with a counter variable called loop_counter. This counter variable is used to control the number of times the repeat group will be carried out. The counter variable is initially set to 1, and every time through the loop, its value is incremented.

```plaintext
constants
define Max_Num = 15 of type integer // maximum number of times through the loop
variables
define loop_counter of type integer // counter variable
begin
set loop_counter = 1 // initial value of counter
while loop_counter < Max_Num do
  increment loop_counter
  display "Value of counter: ", loop_counter
endwhile
...
```

The first time the operations in the repeat group are carried out, the loop counter variable loop_counter has a value equal to 1. The second time through the loop, variable loop_counter has a value equal to 2. The third time through the loop, it has a value 3, and so on. Eventually, the counter variable will have a value equal to the value of Max_Num. When this occurs, the loop terminates.

### 10.3 Repetition with Loop Until

The loop-until construct is similar to the while loop. The main difference is that in the loop-until, the condition is evaluated after the repeat group. This means that the operations in the repeat group, Block1, will be carried out until the condition is true. Figure 9 shows a portion of the flowchart for the loop-until construct.

Independent of the initial evaluation of the condition, the operations in the loop will be carried out at least once. If the condition is true, the
operations in the repeat group will be carried out only once.

The code for the loop-until construct is written with the repeat compound statement, which uses the keywords \texttt{repeat}, \texttt{until}, and \texttt{endrepeat}. The repeat group consists of all operations after the \texttt{repeat} keyword and before the \texttt{until} keyword.

The general concepts of loop counters and loop conditions also apply to the loop-until construct. The following portion of code shows the general structure for the loop-until statement.

\begin{verbatim}
repeat
  \{ statements in Block1 \}
until ( condition )
endrepeat
\end{verbatim}

10.4 Repetition with For Loop

The for-loop is useful when the number of times that the loop is carried out is known in advance. The for-loop explicitly deals with the loop counter. In the \texttt{for} statement, the initial value and the final value of the loop counter...
has to be indicated.

The general structure of the for statement follows. The repeat group consists of the sequence of instructions (written as statements) in Block1.

\[
\text{for } (\text{counter}) = (\text{initial}\_\text{value}) \text{ to } (\text{final}\_\text{value}) \text{ do }
\]

\[
\text{Block1}
\]

\[
\text{endfor}
\]

Every time through the loop, the loop counter is automatically incremented. The last time through the loop, the loop counter has its final value allowed. In other words, when the loop counter reaches its final value, the loop terminates.

The keywords that appear in this statement are: for, to, downto, do, and endfor. The for-loop is similar to the while-loop in that the condition is evaluated before carrying out the operations in the repeat loop.

11 Arrays

There is often the need to declare and use a large number of variables of the same type and carry out the same calculations on each of these variables. Most programming languages provide a mechanism to handle large number of values in a single collection and to refer to each value with an index.

An array is a data structure that can store multiple values of the same type. These values are stored using contiguous memory locations under the same name. The values in the array are known as elements. To access an element in a particular location or slot of the array, an integer value known as index is used; this index represents the relative position of the element in the array; the values of the index start from zero. Figure 10 illustrates the structure of an array with 10 elements.

An array is a static data structure, once the array is declared (and created), its capacity cannot change. An array is created to hold 15 elements, it cannot be changed to hold a larger or smaller number of elements. Using arrays involves three steps:

1. Declaring an array
2. Creating the array
3. Initializing the array, or assigning initial values to the array elements
4. Referencing the elements of the array to access and update the value of the individual elements

11.1 Declaring Arrays

In an array declaration, an identifier is used for the name of the array. The type of the array can be of a simple (primitive) type or a class. The general statement to declare an array of a simple type is:

\[
\text{define } \langle \text{array name} \rangle \text{ array }[] \text{ of type } \langle \text{array type} \rangle
\]

Arrays of simple types must be declared in the variables section for data definitions. For example, the declaration of array temp of type float is:

```plaintext
variables
    define temp array [] of type float
    . . .
```

The declaration of an array of object references, is similar to the dec-
loration of object references. The general OOSimL statement for declaring arrays of object references is:

\[
\text{define } \langle \text{array\_name} \rangle \text{ array } [ ] \text{ of class } \langle \text{class\_name} \rangle
\]

Arrays of object references must be declared in the object references section of data declarations. For example, the declaration of array points of class Point is:

\[
\begin{align*}
\text{object references} \\
\text{define points array } [] \text{ of class Point} \\
\end{align*}
\]

### 11.2 Creating Arrays

After an array has been declared, it should be created and assigned a specified number of elements. The capacity of the array is the number of elements it can hold. The general statement for creating an array is:

\[
\begin{align*}
\text{create } \langle \text{array\_name} \rangle \text{ array } [ \langle \text{capacity} \rangle ] \\
of \text{type } \langle \text{array\_type} \rangle
\end{align*}
\]

The statement for creating the declared array, temp, with capacity of 10 elements is:

\[
\begin{align*}
\text{create temp array } [10] \text{ of type float} \\
\end{align*}
\]

In a similar manner, to create an array of object references, the general statement is:

\[
\begin{align*}
\text{create } \langle \text{array\_name} \rangle \text{ array } [ \langle \text{capacity} \rangle ] \\
of \text{class } \langle \text{class\_name} \rangle
\end{align*}
\]

A convenient and recommended manner to create an array is to use an identifier constant with the value of the capacity of the array. For example, assume the constant MAX_TEMP has a value 10 and NUM_OBJECTS a value of 25, the statements for declaring and creating the array temp and array employees are:

© 2014 J. M. Garrido
constants
    define MAX_TEMP = 10 of type integer
    define NUM_OBJECTS = 25 of type integer
    define NUM = 50 of type integer
variables
    define temp array [] of type float
object references
    define employees array [] of class Employee
    define objservers array[] of class Server
    create temp array [MAX_TEMP] of type float
    create employee array [NUM_OBJECTS] of class Employee
    create objservers array [NUM] of class Server

The declaration of an array indicates the array name and the type of value that
the elements of the array can hold. Creating an array involves creating an array
object with the specified number of elements in the array.

In most practical problems, the number of elements manipulated in the array
is less than the total number of elements. For example, an array is declared with a
capacity of 50 elements but only the first 20 elements of the array are used. Since
the array is a static data structure, elements cannot be inserted or deleted from the
array, only the values of the elements can be updated.

11.3 Referring Individual Elements of an Array

To refer to an individual element in an array, an integer value is used that represents
the relative position of the element in the array. This index value is known as the
index for the array. The index value starts at 0 and the maximum value is the
capacity of the array minus 1.

11.3.1 Arrays of Simple Types

A particular element in the array is denoted by the name of the array followed by
the index value enclosed in rectangular brackets. For example, to assign a value of
342.65 to element 7 of array temp, the statement is:

    set temp[6] = 342.65

The index value can be an integer constant, a constant identifier, or an integer
variable. For example, to refer to element 7 of array temp using variable j as the
index:
11.3.2 Arrays of Object References

An array of object references is not an array of objects since the objects are stored elsewhere in memory. To create an object of class Server and assign its reference to the element with index \( j \) of array \( \text{observers} \), the code is:

\[
\text{create observers}[j] \text{ of class Server}
\]

To invoke a public function of an object referenced in an element of an array, the call statement is used as explained before and the name of the array is followed by the index value enclosed in brackets. For example, to invoke function \( \text{get\_salary} \) of the object referenced by element with index \( j \) in array \( \text{employees} \), the code is:

\[
\text{variables}
\]
\[
\begin{align*}
\text{define obj\_salary of type float} \\
\text{define j of type integer} \\
\text{. . .} \\
\text{set obj\_salary = call get\_salary of employees[j]}
\end{align*}
\]

11.4 Simple Applications of Arrays

This section includes several simple and typical applications of arrays. Class \( \text{Temp} \) is used as an implementation for these discussions. The first part of class \( \text{Temp} \) includes the various data definitions including array \( \text{temp} \). The constructor or initializer method, creates the array.

\[
\text{class Temp is}
\]
\[
\begin{align*}
\text{private} \\
\text{constants} \\
\text{define NUM\_TEMP = 15 of type integer} & \quad // \text{array capacity}
\end{align*}
\]
variables
  define t_index of type integer
  define num_temp_values of type integer // number elements
  define temp array [] of type float
...
public
function initializer is
  create temp array [NUM_TEMP] of type float
...
endfun initializer

11.5 Finding Maximum and Minimum Values in an Array

To find the minimum and/or maximum values stored in an array, all the elements have to be examined. The algorithm for finding the maximum value in informal pseudo-code is:

1. Set the largest value found so far to be the value of the first element of the array.
2. Set the index value of the first element, value 0.
3. Carry out the following steps for each of the other elements of the array
   (a) Examine the value of the next element in the array.
   (b) If the value of the current element is greater than the largest value so far, update the value found so far with this element value and save the index of the element.
4. The result is the index value of the element value found to be the largest in the array.

The algorithm uses two intermediate (or temporary) variables that are used to store the largest value found so far, and the index value of the corresponding element.

Function \texttt{maxtemp} in class \texttt{Temp} implements the algorithm discussed. The function uses array \texttt{temp}, which is declared in the class as an array of type float and that has \texttt{num_temp_values} elements. The function returns the index value with the largest value found.

description
  This function returns the index of the element with the maximum value in the array.
  */
function maxtemp return type integer is
variables
define j of type integer // index variable
// index of element with largest value
define jmax of type integer
define max_val of type float // largest value
begin
set jmax = 0 // index first element
set max_val = temp[0] // max value so far
for j = 1 to num_temp_values - 1 do
  if temp[j] > max_val
  then
    set jmax = j
    set max_val = temp[j]
  endif
endfor
return jmax // result
endfun maxtemp

The minimum value can be found in a similar manner; the only change is the comparison in the if statement.

11.6 Calculating The Average Value in an Array

To find the average value in an array, all the elements have to be added to an accumulator variable, sum. The algorithm for computing the average value in informal pseudo-code is:

1. Set the accumulator variable, sum, to the value of the first element of the array.
2. For each of the other elements of the array, add the value of the next element in the array to the accumulator variable.
3. Divide the value of the accumulator variable by the number of elements in the array. This is the result value calculated.

The algorithm uses an accumulator variable that is used to store the summation of the element values in the array. In an array \( x \) with \( n \) elements. The summation of \( x \) with index \( j \) starting with \( j = 1 \) to \( j = n \) is expressed mathematically as:

\[
sum = \sum_{j=1}^{n} x_j.
\]

The average is calculated simply as \( sum/n \). Function average_temp in class Temp implements the algorithm discussed. The function uses array temp, which
is declared in class Temp as an array of type float and that has num_temp_values elements. The function returns the average value calculated. The code for the function follows.

description
This function computes the average value of the array temp. The accumulator variable sum stores the summation of the element values. */

function average_temp return type float is
variables
  define sum of type float // variable for summation
  define ave of type float   // average value
  define j of type integer
begin
  set sum = 0
  for j = 0 to num_temp_values - 1 do
    add temp[j] to sum
  endfor
  set ave = sum / num_temp_values
  return ave
endfun average_temp

11.7 Array Parameters

A function can define a parameter as an array. The array can then be pass as argument when calling the function. A copy of the array is not actually passed as an argument, only a reference of the array is passed.

To define a function with an array as a parameter, the header of the function definition must indicate this with the parameters keyword. The general syntax for a function header that includes a parameter definition is:

description
  . . .
  */
  function ⟨ function_name ⟩ parameters
  ⟨ parameter_list ⟩ is
  . . .
endfun ⟨ function_name ⟩

The parameter_list part of the function header can include one or more array definitions. The array name is followed by at least one empty pair of brackets.
A function definition that finds the minimum value in an array of type float defines the array as a parameter and an integer parameter. This second parameter is the number of elements in the array. The function returns the minimum value. The complete definition of function minimum follows.

```
description
This function calculates the minimum value of
an array parameter tarray, it then returns the
result.
*/
function minimum return type float parameters
tarray array [] of type float,
umel of type integer is
variables
    define min of type float // local variable
define j of type integer
begin
    set min = tarray[0]
    for j = 1 to numel - 1 do
        if min < tarray[j] then
            set min = tarray[j]
        endif
    endfor
    return min
endfun minimum
```

To call a function and pass an array as an argument, only the name of the array is used. For example, the following code calls function minimum and pass array mtime.

```
constants
define KNUM = 100 of type integer
variables
define mtime array [] of type float
define mintime of type float
    . . .
create mtime array [KNUM] of type float
    . . .
set mintime = call minimum using mtime, KNUM
    . . .
```
The above code with call statement is included in another function of the same class. The call to function \textit{minimum} is carried out in an assignment statement because the function returns a value that is assigned to variable \textit{mintime}.

11.8 Arrays with Multiple Dimensions

Arrays with more than one dimension can be defined to solve mathematical problems with matrices, computer games, and so on. Two-dimension arrays are easier to understand and matrices are the most common type of problems. These are mathematical structures with values arranged in columns and rows. Two index values are required, one for the rows and one for the columns.

To define a two dimensional array, two numbers are defined each in a pair of brackets. The first number defines the range of values for the first index (for rows) and the second number defines the range of values for the second index (for the columns).

For example, the statements to declare and create a two-dimensional array named \textit{matrix} with capacity 15 rows and 20 columns, the code is:

\begin{verbatim}
constants
define ROWS = 15 of type integer
define COLS = 20 of type integer
variables
define matrix array [][] of type float
...create matrix array [ROWS][COLS] of type float
\end{verbatim}

To reference the elements of a two-dimensional array, two indices are required. For example, the following code sets all the elements of array \textit{matrix} to 0.0:

\begin{verbatim}
for j = 0 to COLS - 1 do
  for i = 0 to ROWS - 1 do
    set matrix [i][j] = 0.0
  endfor
endfor
\end{verbatim}

For this initialization of array \textit{matrix}, two loop definitions are needed, an outer loop and an inner loop (this is also known nested loops). The inner loop varies the row index and outer loop varies the row index. The assignment statement sets the value 0.0 to the element at row \textit{i} and column \textit{j}.
12 Strings

Strings are special arrays with data of type character. Recall that type character is a primitive type. A string is a sequence of data items of type character. Most of the data manipulated by programs is either numeric or string data.

This section explains and discusses the concepts related to strings and their manipulation. There are several operators that are special to strings. These string operators are explained and applied in a few examples presented in the section. The declaration and manipulation of strings in Java, is briefly discussed.

12.1 Declaring Strings

Strings variables are of type string and can be declared with or without their values, in a similar manner to the variables of primitive types. The following declares a string variable, message.

\[
\text{define message of type string}
\]

A value of type string is a string constant enclosed in quotes. For example, the following statement assigns a string value to the string variable message.

\[
\text{set message = "Hello, world!"}
\]

A string variable can be declared with a value, the following declares string variable s2 with the string constant "Hi, everyone!".

\[
\text{define s2 = "Hi, everyone" of type string}
\]

The two string variables declared above can be displayed on the console with the following statements:

\[
\text{display message}
\]
\[
\text{display s2}
\]

When executing the program that includes these two statements (discussed above), the following message will appear on the screen:

Hello, world!
Hi, everyone! 
12.2 String as an Array

A string value can be thought of as a sequence of data items of type `character` and implemented as a special array of characters. Recall that type `character` is a primitive type. This section discusses additional string operators for string manipulation.

Strings are special variables, they are immutable. After a string has been given an assigned string value, it cannot be changed.

12.2.1 Length of a String

Every string has a different number of characters. The string operator `length` gets the number of characters in a string variable. This operator is normally used in an assignment statement and the target variable should be of type `integer`. The general syntax for the assignment statement with the `length` operator follows.

\[
\text{set } \langle \text{int}_\text{var} \rangle = \text{length of } \langle \text{str}_\text{var} \rangle
\]

For example, to get the number of characters in string variable `message` and assign this to integer variable `num`, the complete statement is:

\[
\text{set num = length of message}
\]

12.2.2 Retrieving a Character of a String

To get a copy of a character located at some specified relative position of the string, the operator `charat` is used. The relative position of a character is known as the index. The integer value of the index has a range from 0 to the value `length - 1`.

The `charat` operator is normally used in an assignment statement and the target variable should be of type `character`. The general syntax for the assignment statement with the `charat` operator follows.

\[
\text{set } \langle \text{char}_\text{var} \rangle = \text{charat } \langle \text{index} \rangle \text{ of } \langle \text{str}_\text{var} \rangle
\]

For example, to get the character at the index value 7 in string variable `message` and assign this to character variable `llchar`, the portion of code is:

```plaintext
variables
    define llchar of type character
...
set llchar = charat 7 of message
```
When this statement executes, the value of variable *llchar* becomes ‘W’, which is the character at index position 7 of the string variable *message*.

### 12.3 Finding the Position of a Character

To find the position of a character within a string is finding the value of the index for the given character. The *indexof* operator searches the string from left to right. This operator is also used in an assignment statement. The general structure of this statement with the *indexof* operator follows.

```plaintext
set ⟨ int_var ⟩ = indexof ⟨ char_var ⟩ of ⟨ str_var ⟩
```

For example, to get the index value in string variable *message* for the character ‘r’ and assign this to integer variable *num*, the complete statement is:

```plaintext
variables
define num of type integer
...
set num = indexof 'r' of message
```

When this statement executes, the value of variable *num* becomes 9. If the starting index value is known, it can be included after the character in the assignment statement. If the character indicated does not exist in the string then the value assigned is −1.

### 12.4 Retrieving a Substring from a String

A substring is part of a string. To retrieve a substring from a string, the index position of the character that starts the substring is needed. By default, the end of the substring is also the end of the string. The *substring* operator is used in an assignment statement and gets a substring that starts at a given index position up to the end of the string. The variable that receives this value is a string variable. The general structure of the assignment statement with the *substring* operator follows.

```plaintext
set ⟨ str_var1 ⟩ = substring ⟨ index ⟩ of ⟨ str_var2 ⟩
```

For example, to get the substring value that starts at index position 7 in string variable *message* and assign this to string variable *yystr*, the portion of code is:

```plaintext
variables
define yystr of type string
define num = 7 of type integer
...
set yystr = substring num of message
```
When this statement executes, the value of variable *yystr* becomes "World!". Two index values are used for substrings that have start and end index positions. For example, to retrieve the substring located at index positions 8 to 10 of string variable *message*, the portion of code with the assignment statement follows.
variables
  define yystr of type string
  define num1 = 8 of type integer
  define num2 = 10 of type integer
  ...
  set yystr = substring num1 num2 of message

When this statement executes, the value of variable yystr becomes "or1".

12.5 Finding the Position of a Substring

To get the position of a substring within a string is very similar to that of a character. The operator indexof is also used to search for substrings. This operator searches the string from left to right. This operator is also used in an assignment statement. The general structure of this statement with the indexof operator follows.

\[
\text{set } \langle \text{int var} \rangle = \text{indexof} \langle \text{str var1} \rangle \text{ of } \langle \text{str var2} \rangle
\]

For example, to get the index value in string variable message for the substring "llo" and assign this to integer variable num, the complete statement is:

variables
  define num of type integer
  define mystr = "llo" of type string
  ...
  set num = indexof mystr of message

When this statement executes, the value of variable num becomes 9. If the starting index value is known, it can be included after the substring in the assignment statement. If the substring indicated does not exist in the string then the value assigned is \(-1\).

12.6 Joining Two or More Strings

Two or more strings can be joined one after the other to build a larger string. The operation of joining two strings together is called concatenation. The concat operator is used to join two strings. This operator must be part of an assignment statement and appears between two variables. The following presents the general syntax of the assignment statement with the concat operator, variables var_2 and var_3 are joined the second after the first. The result of the concat operation is assigned to variable var_1.

\[
\text{set } \langle \text{var}_1 \rangle = \langle \text{var}_2 \rangle \text{ concat } \langle \text{var}_3 \rangle
\]
For example, the following statement joins together the string variable \textit{message}, the string constant " and ", and the string variable \textit{s2}. The resulting string is assigned to string variable \textit{s3}.

\begin{verbatim}
string s3
...
set s3 = message concat " and " concat s2
\end{verbatim}

When this statement executes, the value of string variable \textit{s3} becomes:

"hello World! and Hi, everyone!"

The flexibility of the operation that joins two or more strings is that a string variable can be concatenated to a variable of a different primitive type. For example, in the following statement string variable \textit{s3} is joined with a blank space and joined with the integer variable \textit{j} and assigned to string variable \textit{s4}.

\begin{verbatim}
define j of type integer
define s4 of type string
....
set s4 = s3 concat " " concat j
\end{verbatim}

The numeric value of variable \textit{j} is converted automatically to a string then joined with the other string values. The blank space was joined between variables \textit{s3} and \textit{j} for appearance when the string variable \textit{s4} is displayed on the console.

\subsection{12.7 Comparing Strings}

In OOSimL, a string \textit{s1} can be compared with another string \textit{s2} using the \texttt{equals} operator. This operator compares two strings and evaluates to a truth-value so it can be used with an \texttt{if} statement. The general syntax for this operator follows, variable \texttt{str.var1} is compared to \texttt{str.var2}.

\begin{verbatim}
if ( str.var1 ) equals ( str.var2 )
\end{verbatim}

For example, the following statement tests if string \textit{s1} is equal to string \textit{s2}:

\begin{verbatim}
variables
define s1 of type string
define s2 of type string
\end{verbatim}
begin
  if s1 equals s2
  then

Another operator for string comparison is the compareto operator, which compares two strings and evaluates to an integer value. If this integer value is equal, the two strings are equal. If the integer value is greater than zero, the first string is higher alphabetically with respect to the second string. If the integer value is less than zero, the first string is lower alphabetically with respect to the second string. The compareto operator is used in an assignment statement. The general structure of an assignment statement with this operator follows.

```
set { int_var } = { str_var1 } compareto { str_var2 }
```

The following code compares string variables s1 and s2 in an assignment statement with integer variable test. This integer variable is then tested in an if statement for various possible sets of instructions.

variables
define s1 of type string
define s2 of type string
define test of type integer  // result of string comparison

begin
  if test = 0
  then
    if test > 0
    then
      else

In the example, variable test holds the integer values with the result of the string comparison of s1 and s2. Variable test is subsequently evaluated in an if statement for further processing.
13 Inheritance

The previous sections discussed class relationships. The two basic categories of relationship among classes are composition and inheritance. These relationships are modeled in UML diagrams and it can be said that composition is a horizontal relationship and inheritance is a vertical relationship.

Inheritance is a mechanism provided by an object-oriented language for defining new classes from existing classes. This section explains the basic inheritance relationships and their applications in some detail. Inheritance enhances class reuse, i.e., the use of a class in more than one application.

13.1 Classification

Classification is a modeling concept and for a given application, it refers to the grouping of the objects with common characteristics. In this activity, the classes and their relationships are identified. Some classes of an application are completely independent, only the class with function main has a relationship with them. The other classes in the application are related in some manner and they form a hierarchy of classes.

In a class hierarchy, the most general class is placed at the top. This is the parent class and is also known as the super-class (or the base class). A derived class inherits the characteristics (all attributes and operations) of its parent class. A derived class can be further inherited to lower-level classes. In the UML class diagram, an arrow with an empty head points from a subclass (the derived class) to its base class to show that it is inheriting the features of its base class. Since in the UML diagram, the arrow showing this relationship points from the subclass up to the base class, inheritance is seen as a vertical relationship between two classes (see Figure 11).

13.2 Base Class

Inheritance is a mechanism by which a new class acquires all the non-private features of an existing class. This mechanism is provided by object-oriented programming languages. The existing class is known as the parent class, the base class, or the super class. The new class being defined, which inherits the non-private features of the base class, is known as the derived class or subclass.

The new class acquires all the features of the existing base class, which is a more general class. This new class can be tailored in several ways by the programmer by adding more features or modifying some of the inherited features.

The main advantage of inheritance is that the definition and development of a class takes much less time than if the class were developed from scratch. Another advantage of inheritance is that it enhances class reuse. A subclass can be:

- An extension of the base class, if it includes its own attributes and operations, in addition to the derived characteristics it inherits from the base class.
• A specialized version of the base class, if it overrides (redefines) one or more of the characteristics inherited from its parent class.

• A combination of an extension and a specialization of the base class.

When a class inherits the characteristics from more than one parent class, the mechanism is called multiple inheritance. Most object-oriented programming languages support multiple inheritance, OOSimL supports only single inheritance. Therefore, in modeling, it is useful to show this in the class diagrams.

In UML terminology, generalization is the association between a general class and a more specialized class or extended class. This association is also known as inheritance and it is an important relationship between classes. In the UML class diagram, the arrow that represents this relationship points from a class (the derived class) to its parent class.

Figure 11: An inheritance relationship.

Figure 11 illustrates a simple class hierarchy with inheritance. The parent class is Polygon and the subclasses are: Triangle, Rectangle, and Parallelogram that inherit the features from the parent class. The idea of a subclass and a subtype is important. All objects of class Parallelogram are also objects of class Polygon, because this is base class for the other classes. On the contrary, not all objects of class Polygon are objects of class Parallelogram.

13.3 Defining new Classes with Inheritance

In OOSimL, the subclass acquires all the public and protected features of the base class. A protected feature is only accessible to the class that defines it and to the subclasses. The only public features that are not inherited by the subclasses are the initializer functions of the base classes.

The definition of a subclass must include the keyword inherits to name the base class that it inherits. The general structure of the statement that defines a subclass is:
Since a subclass can also be inherited by another class, it often includes protected features in addition to the private and public ones. In UML class diagrams, a feature of the class is indicated with a plus (+) sign if it is public; with a minus (-) sign if it is private; and with a pound (#) sign if it is protected.

### 13.4 Initializer Functions in Subclasses

An initializer function of the subclass will normally need to invoke the initializer function of the base class. The special name given to the initializer function of the base class is `super`. This call must be the first statement in the initializer function of the subclass. Calling this function may require arguments, and these must correspond to the parameters defined in the initializer function of the base class.

Initializer (constructor) functions are not inherited. A derived class must provide its own initializer functions. These are the only public features that are not inherited by the subclass. The statement to call or invoke an initializer function of the base class from the subclass is:

```plaintext
call super (using argument_list)
```

If the argument list is absent in the call, the initializer function invoked is the default initializer of the base class.

Suppose a new class, `Toyball`, is defined that inherits the features of an existing class `Ball`, the base class. The attributes of class `Ball` are `color` and `size`. The color attribute is coded as integer values (white is 0, blue is 2, yellow is 3, red is 4, black is 5). Class `Ball` includes an initializer function that sets initial values to these two attributes.

The subclass `Toyball` has one other attribute: `weight`. The initializer function of this class needs to set initial values to the three attributes, two attributes of the base class and the one attribute of the subclass. The two attributes of the base class (`Ball`) are set by invoking the function `super` in the initializer function of the subclass, `Toyball`.

```plaintext
class Toyball inherits Ball is
```

© 2014 J. M. Garrido
The attributes of a class are private, so the only way to set the initial values for the attributes of base class is to invoke its initializer function of the base class. In OOSimL, this is accomplished with the statement:

call super using icolor, isize

14 Abstract Classes

As discussed previously, inheritance is a mechanism provided by an object-oriented language for defining new classes from existing classes. Inheritance enhances class reuse, i.e., use of a class in more than one application.

Inheritance also helps in dealing with generic modeling and programming. With generics, the more general classes are separated from the more specific or concrete classes. This separation helps to enhance re-use of the more general definitions in modeling and in programming.

Abstract classes help improve the object-oriented model of the problem. They help clarify the understanding of the model and provide good specifications. Interfaces allow the introduction to pure specifications, the complete separation between specification and implementation. Polymorphism is a mechanism that allows more flexibility in the design and provides generic programming.

This section discusses the three important and related concepts mentioned above: abstract classes, interfaces, and polymorphism.
14.1 Abstract Classes

Figure 12 illustrates a simple inheritance relationship with four classes. These are: 
Gfigures, Triangle, Circle, and Rectangle.

![Class Diagram]

Figure 12: A generic base class.

The characteristics of the geometric figures are related in some way. Assume that the relevant attributes defined in these classes are \textit{height}, \textit{base}, and \textit{radius}. The relevant functions defined in the classes are \textit{area} and \textit{perimeter}.

An abstract class is one that includes one or more abstract methods. An abstract method has no implementation; only has the class declaration (also known as the class specification).

An abstract class cannot be instantiated; it is generally used as a base class. The subclasses override the abstract methods inherited from the abstract base class.

14.2 Defining an Abstract Class

To define an abstract class, the keyword \texttt{abstract} should be used before the keyword \texttt{class}. Every function that does not include its implementation is also preceded by the keyword \texttt{abstract}. An abstract class definition has the following the general structure:

\begin{verbatim}
  description
  abstract class \{ class_name \} is
  private
    // private attributes
  constants
    . . .
  variables
    . . .
  objects
\end{verbatim}
The attributes of derived classes, *Rectangle*, *Triangle*, and *Circle* are different. Classes *Rectangle* and *Triangle* have attributes *height* and *base*; Class *Circle* has attribute *radius*. The calculations of area and perimeter are different in every one of these subclasses. In this situation, the base class (**Gfigures**) cannot include the implementations for functions *area* and *perimeter*. It can only provide the prototypes for these functions.

A base class is an abstract class when it does not provide the implementation for one or more functions. Such base classes provide single general description for the common functionality and structure of its subclasses. An abstract class is a foundation on which to define subclasses. Classes that are not abstract classes are known as concrete classes.

The base class **Gfigures** is an abstract class because it does not provide the implementation of the functions *area* and *perimeter*. The code with the definition of class **Gfigures** is as follows.

description
   This simple abstract class has two functions.
/*
abstract class Gfigures is
   public
description
   This function computes and returns the area
   of the geometric figure.
   */
abstract function area of type double
description
   This function computes and returns the perimeter
   of the geometric figure.
   */
abstract function perimeter of type double
endclass Gfigures

Since the abstract class **Gfigures** does not include the body of the functions *area* and *perimeter*, objects of this class cannot be created. In other words, class **Gfigures** cannot be instantiated.
15 Interfaces

An interface is similar to a pure abstract class. It does not include attribute definitions and all its methods are abstract methods. Constant definitions are allowed. An interface does not include constructors and cannot be instantiated.

To define an interface, the keyword `interface` is used instead of `abstract class` and before the name of the interface. For the functions, the keyword `abstract` is not needed because all the functions are abstract functions. In a similar manner, all features of an interface are implicitly public. An interface definition has the following the general structure:

```plaintext
description
interface (interface_name) is
  public
  constants
  // public operations
endinterface (interface_name)
```

The following interface, named `Iball`, defines the specification for the behavior of objects of any class that implements this interface. Note that the structure of the interface is very similar to that of an abstract class.

```plaintext
description
  This is a simple interface
Jan 2003, J Garrido
*/
interface Iball is
  public
  // public methods
description
  This method accesses the value of attribute
  color  */
function get_color of type integer
  //
description
  This method reads the value of attribute color
  from the console
  */
function get_size of type real
  //
```

© 2014  J. M. Garrido
description
   This function returns the value of the
   move_status.
   */
function get_m_status of type character
description
   This function displays the color, status, and
   size of the object.
   */
function show_state
description
   This function changes the move_status of
   the object to move.   */
function move
description
   This function changes the move_status of
   the object to stop.   */
function stop
endinterface Iball

15.1 Using an Interface

A class makes use of an interface by implementing it. All methods declared in the
interface must be implemented in the class that implements it. This class can define
additional features. The class that implements an interface must use statement
implements. The header for the class definition that uses this statement has the
following the general structure:

description
   . . .
class ( cls_name ) implements ( interface_name ) is
   . . .
endclass ( cls_name )

15.2 Subtypes

Although an interface and an abstract class cannot be instantiated, both can be
used as super types for object references. This implies that interfaces and abstract
classes are useful for declaring object references.

For example, refer again to Figure 12. Class Gfigures is an abstract class
and the other classes are subclasses. An object reference can be declared of type
Gfigures.
object references
define gen_figure of class Gfigures
...

In a similar manner, objects of the subclasses can be declared of each of their
classes. The subclasses are considered subtypes of type \texttt{Gfigures}. For example,
the following declaration defines three object references, \texttt{triangle_obj}, \texttt{circle_obj},
and \texttt{rec_obj}.

object references
define triangle_obj of class Triangle
define circle_obj of class Circle
define rect_obj of class Rectangle
...

The types of these object references declared are considered subtypes in the
problem domain because of the original class hierarchy represented in Figure 12.
For this organizational structure of the problem, the object reference \texttt{triangle_obj}
is declared of type \texttt{Triangle}, but is also of type \texttt{Gfigures}. In fact, any object
reference of type \texttt{Triangle} is also of type \texttt{Gfigures}. The same principle applies to
the object references declared with the types \texttt{Circle} and \texttt{Rectangle}. Of course,
the opposite is not true; any object reference of type \texttt{Gfigures} is not also of type
\texttt{Rectangle}, \texttt{Circle}, or \texttt{Triangle}.

An interface can also be used as a super type and all the classes that implement
the interface are considered subtypes.

16 Polymorphism

An object reference of a super type can refer to objects of different subtypes. This
is possible from the subtyping principle explained before.

16.1 Simple Application of Polymorphism

To illustrate this concept, the following code creates objects for the object references
\texttt{triangle_obj}, \texttt{circle_obj}, and \texttt{rect_obj}. Assume there is a declaration for variables \texttt{x},
\texttt{y}, \texttt{z}, and \texttt{r}.

create triangle_obj of class Triangle using \texttt{x}, \texttt{y}, \texttt{z}
create circle_obj of class Circle using \texttt{r}
create rect_obj of class Rectangle using \texttt{x}, \texttt{y}
The object reference \textit{gen}\	extit{figure} of class \textit{Gfigures} can be assigned to refer to any of the three objects \textit{triangle}\	extit{obj}, \textit{circle}\	extit{obj}, and \textit{rect}\	extit{obj} created previously. For example, the following statement implements such an assignment.

\begin{verbatim}
set gen_figure = triangle_obj
\end{verbatim}

This is perfectly legal because the type of object reference \textit{triangle}\	extit{obj} is a subtype of the type of object \textit{gen}\	extit{figure}. After this assignment, it is possible to invoke a function of the abstract class \textit{Gfigures} that is implemented in the subclass \textit{Triangle}. For example, the following code invokes function \textit{perimeter}:

\begin{verbatim}
call perimeter of gen_figure
\end{verbatim}

The actual function invoked is the one implemented in class \textit{Triangle} because it is the type of object reference \textit{triangle}\	extit{obj}. At some other point in the program (possibly in function \textit{main}), another similar assignment could be included. The following code assigns the object reference \textit{circle}\	extit{obj} to the object reference \textit{gen}\	extit{figure}.

\begin{verbatim}
set gen_figure = circle_obj
\end{verbatim}

The call to function \textit{perimeter} is the same as before, because the three subtypes represented by the three subclasses \textit{Rectangle}, \textit{Circle}, and \textit{Triangle} implement function \textit{perimeter}. Since the implementation for this function is different in the three classes, the runtime system of the Java compiler selects the right version of the function.

Polymorphism is a runtime mechanism of the language that allows the selection of the right version of a function to be executed depending on the actual type of the object. Only one among several possible functions is really called. This function selection is based on \textit{late binding} because it occurs at execution time.

\subsection*{16.2 Heterogeneous Array}

The following problem applies the concepts of arrays and of polymorphism. A list of objects is required to store geometric figures and to calculate the perimeter and area of each figure when selected from the list. The geometric figures are circles, triangles, and rectangles.

The list is designed as an array of objects of class \textit{Gfigures}. The solution to this problem is generic by using this abstract class, the base class for all the other classes. The type of the array is \textit{Gfigures}, this class was defined previously. The capacity of the array is set with constant \textit{MAX GEOM}.  

© 2014 J. M. Garrido
Each element of the array is an object reference of a concrete class, \texttt{Circle}, \texttt{Triangle}, or \texttt{Rectangle}. An object is created for every object reference in the array. To process the array, the index value is used to access the needed element and invoke the methods to compute the perimeter and the area of the geometric figure. The execution of these functions is carried out via polymorphism. The array of geometric figures is declared as:

\begin{verbatim}
object fig_list array [MAX_GEOM] of class Gfigures
\end{verbatim}

The type of the array is the base class \texttt{Gfigures}, an abstract class. The types of the objects referred by the array elements are of the concrete classes that are subtypes of \texttt{Gfigures}.

\section{17 Basic Graphical User Interfaces}

In console applications, all input/output is carried out only through the console (video screen and keyboard). With graphical user interfaces (GUIs), the users have a much better way to interact with the programs. The user actually interacts with graphical elements, such as buttons, dialog boxes, menus, and so on.

Programs with graphical user interfaces present the user with a nice and convenient arrangement of graphical components and guide the user in a very effective manner when interacting with the application.

OOSimL provides access to all Java graphical libraries (or packages) of classes. The basic class packages are the abstract windows toolkit (\texttt{AWT}) and \texttt{Swing}; it is an enhanced collection of graphical classes.

This section discusses and explains the design and construction of simple GUIs using the two graphical libraries. Most of the examples presented use mainly the \texttt{Swing} library. Also discussed in this section are \textit{applets}, which are small applications that execute in a Web browser.

\subsection*{17.1 Graphical Objects}

When running GUI applications, a typical user looks at some graphical objects on the screen. Some of these objects display data of the program. Other graphical objects allow the user to enter data to the program. The user can also point and click with other graphical objects, and in this way, can interact with the program in execution. The most relevant objects that form part of a graphical user interface are:

- Containers
- Components
- Events
- Listeners
17.2 Components and Containers

Simple GUIs have only containers and components. Graphical interfaces that only include these objects present very limited user interaction. Adding events and listeners to components and containers, the full advantage of user interaction is made available.

A container is an object that can hold graphical components and smaller containers. A container object also allows its component objects to be arranged in various ways. This arrangement of objects is facilitated by special objects called layout managers. Examples of container objects are frames and panels.

A component is a small graphical object that displays data, allows the user to enter data, or simply indicates some condition to the user. Component objects are usually arranged with other components in a container object. Examples of component objects are buttons, labels, and text fields.

A simple graphical user interface is shown in Figure 13. The frame is divided into two panels, and each panel contains several graphical components.

![Figure 13: General structure of a GUI.](image)

17.3 Using Graphic Libraries

A package is a group of related classes. A library package is a predefined group of classes that are available on the current environment.

The use statement is required by any class definition that needs access to one or more classes in a library package. Most programs that include GUI, need to access the two Java class packages, AWT and Swing. In the programs, a use statement must be included at the top of a class definition.

The following two lines of code are the ones normally required by programs that access the graphical libraries AWT and Swing. The import statements shown give access to all the classes in the two packages.
17.4 Frames

The largest type of container is a frame. This container can be created simply as an object of class JFrame of the Swing library.

An empty frame window can be built by creating an object of class JFrame with a specified title and setting the size for it. The relevant properties of the frame are its title, color, and size.

Class Frame_sample implements the construction of an empty frame with title “Frame sample” and size of 400 by 300 pixels. The title is set by invoking the constructor of the frame object, and the size is set by invoking the method setSize of object frame_obj. To make the frame window visible, method setVisible is invoked.

```java
use all javax.swing // Library for graphics
description
This class creates and displays an empty frame window. */
class Frame_sample is
public
description
This is the main function of the application. */
function main is
constants

define WIDTH = 400 of type integer
define HEIGHT = 300 of type integer
object references
define frame_obj of class JFrame
begin
create frame_obj of class JFrame using "Frame sample"
call setSize of frame_obj using WIDTH, HEIGHT
call setVisible of frame_obj using true
endfun main
endclass Frame_sample
```

The size of the graphical object on the screen is measured in pixels. A pixel is the smallest unit of space that can be displayed on the video screen. The total
number of pixels on the screen defines the resolution of the screen. High-resolution screens have a larger number of pixels, and the pixels are much smaller.

17.5 Simple Components

The simplest types of components are labels, which can display text titles and images. Text labels just display their text titles when they appear on a container of a window. A text label is defined as an object of class JLabel. The following statements declare two text labels:

```java
define blabel1 of class JLabel // text label
define blabel2 of class JLabel
```

When the objects of class JLabel are created, their text titles are defined. The following statements create the two text objects and define their associated text titles.

```java
create blabel1 of class JLabel using "Kennesaw Java Preprocessor"
create blabel2 of class JLabel using "The Language for OOP"
```

Labels can also display pictures by indicating icons for the pictures. Image labels display a picture by indicating the corresponding icon. In classes using Swing, a picture is set up into an icon in a label, so that the classes can position the label in a container and display it. The pictures are normally in a standard format, such as JPEG or GIF.

A picture is defined as an icon, which is an object of class ImageIcon. The icon is then defined as part of the label. The following statements declare an object variable of class ImageIcon and an object of class JLabel.

```java
define oosimlimage of class ImageIcon // image
define oosimllabel of class JLabel // for image
```

The icon object is created with a picture stored in a specified picture file. The following statement creates the icon object oosimlimage with a picture in the file oosiml.gif.

```java
create oosimlimage of class ImageIcon using "oosiml.gif"
```
Finally, with the icon object created, it can be defined as part of a label. The following statement creates the label object with the icon defined in object variable `oosimlimage`.

```java
create oosimlabel of class JLabel using oosimlimage
```

### 17.6 Adding Components to a Window

Components cannot be added directly to a window, which is an object of class `JFrame`. A special container called the `content pane` defines the working area for the window. All the graphical elements, components, and smaller containers are added to the content pane. The content pane is an object of class `Container`.

There are several ways to arrange graphical elements in the content pane. The type of arrangement is defined by the layout manager selected. The `AWT` package provides six layout managers:

- **Border**, which arranges the components in the north, east, west, center, and south positions in the container
- **Flow**, which arranges the components in the container from left to right
- **Grid**, which arranges the components in a matrix with row and column positions
- **Box**, which arranges the components in a single row or single column
- **Card**, which arranges the components in a similar manner as the stack of cards
- **Gridbag**, which arranges the components in a similar manner as the grid but with variable size cells

The most common layout managers are the first three: border, flow, and grid. Figure 14 shows the positioning of components using the border layout manager.

The content pane is an object of class `Container`. To manipulate the content pane object, a reference to this object is accessed by invoking method `getContentPane` of the window. Before adding the various graphical elements to the content pane, the layout must be set by invoking the method `setLayout` of the content pane.

The following class, `Oosimlogo`, sets up a window (an object of class `JFrame`) with three components: two text labels and an image label.

```java
use all javax.swing       // Library for graphics
use all java.awt

description
    This class creates and displays a frame window
    with an image and two text labels. */
```

© 2014 J. M. Garrido
class Oosimlogo is public
description
This is the main function of the application. */
function main is
  constants
    define WIDTH = 400 of type integer
    define HEIGHT = 300 of type integer
  object references
    define frame_obj of class JFrame // window
    // content pane
    define cpane of class Container
    define blabel1 of class JLabel // text label
    define blabel2 of class JLabel
    define myimage of class ImageIcon // image
    define mylabel of class JLabel // for image
    // layout manager
    define lmanager of class BorderLayout
begin
  create frame_obj of class JFrame using "OOSimL logo"
  create blabel1 of class JLabel using "Kennesaw Java Preprocessor"
  create blabel2 of class JLabel using "The Language for OO Simulation"
create myimage of class ImageIcon using "oosiml.gif"
create mylabel of class JLabel using myimage
create lmanager of class BorderLayout
call getContentPane of frame_obj return to cpane
call setLayout of cpane using lmanager
// add the text image label and text label
// to the content pane of the window
call add of cpane using mylabel,
    BorderLayout.CENTER
call add of cpane using blabel1,
    BorderLayout.NORTH
call add of cpane using blabel2,
    BorderLayout.SOUTH
call setSize of frame_obj using WIDTH, HEIGHT
call setVisible of frame_obj using true
endfun main
endclass Oosimlogo

Figure ?? shows the window that appears on the screen when the program in class Oosimlogo executes.

Figure 15: A frame with three components.
17.7 Attributes of Frames

Two attributes of frames that are used in the previous examples are title and size. The title was set with the constructor of class JFrame when creating a frame object. This attribute can be set at any time by invoking method setTitle with a string argument. For example, the following statement sets a title to the frame object declared with reference frame_obj in the previous example:

\[
\text{call setTitle using } "\text{New Frame Title}" \]

The attributes of the frame object are normally set before displaying the frame on the screen. As mentioned before, the units for size are in pixels. The previous example used two named constants: WIDTH and HEIGHT. This is the recommended manner to set the values for the size. The size is set by invoking method setSize with the values in pixels for width and height of the frame. To set the size of the frame referenced by frame_obj, the statement is:

\[
\text{call setSize of frame_obj using WIDTH, HEIGHT} \]

Another attribute of a frame is the color. This attribute is normally set to a container in the frame, such as the content pane of the frame or the panels defined and contained in the content pane. In other words, the color of any of these containers can directly be set. The most common method to invoke for a container is setBackground. The argument is normally a predefined constant in class Color, which is available in the AWT package. These constants represent various colors to apply as background color to the container. The most common constants for the colors are listed in Table 1.

<table>
<thead>
<tr>
<th>Color</th>
<th>Color</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>orange</td>
<td>green</td>
</tr>
<tr>
<td>gray</td>
<td>magenta</td>
<td>pink</td>
</tr>
<tr>
<td>red</td>
<td>white</td>
<td>yellow</td>
</tr>
<tr>
<td>darkGray</td>
<td>lightGray</td>
<td></td>
</tr>
</tbody>
</table>

Method setBackground is invoked with the selected color to set the background color of a container in a frame object. This method is defined in class JFrame. The following statement sets the background color to pink in frame frame_obj of the previous example. Recall that cpane is the content pane of frame_obj.

\[
\text{call setBackground of cpane using Color.pink} \]
17.8 Events and Listeners

An event is a special object that represents an occurrence or signal. This occurrence happened during the interaction between the user and the executing program. Usually, the most important events are the ones originated by some user action.

Examples of user actions that originate events are the click of the mouse, starting of mouse movement, a keystroke on the keyboard, and so on.

Each of these actions generates a specific type of event. A program that is dependent on events while it executes is called event-driven, because the behavior of the program depends on the events generated by user actions.

A listener is an object that waits for a specific event to occur and responds to the event in some manner. Each listener has a relationship with an event or with a group of similar events. Listener objects must be carefully designed to respond to its type of events.

17.9 Objects that Generate Events

Buttons and text fields are examples of graphical components that generate events as a result of user actions. The listener object will respond when the user clicks on the button. The behavior of the listener object should be defined in a class. For a button, in addition to declaring and creating the button as an object of class JButton, the declaration and creation of the listener object is also required.

When a user clicks a button, the type of event generated by the button is called an action event. Creating the action event and invoking the appropriate function of the listener object to handle the event is done automatically.

The action event generated by the button is sent to a corresponding listener object that responds to the event. For this, the listener object must be set to respond to an event generated by the button object—this is known as registering the listener object with the button object. The sequence of steps to set up a button is:

1. Define the class that implements the behavior of the listener object.
2. Declare the button object variable.
3. Declare the listener object for the button.
4. Create the button object with a title.
5. Create the listener object that will respond to the action event generated by the button.
6. Register the listener object for the button.
7. Add the button object to the container.

The AWT and Swing packages provide several classes and interfaces for defining listener objects. For action events, the interface ActionListener must be implemented by the class that defines the behavior of the listener object.
The button object generates an action event that it sends to the listener object by invoking method `actionPerformed`. The interaction among these objects occurs automatically (behind the scenes).

### 17.10 Adding a Button to a Window

As mentioned earlier, a button is an object of class `JButton`. The following statements declare an object reference of class `JButton`.

```java
define mybutton of class JButton
```

The button object is created with a title. The following statement creates the button object with the title “Push to Quit.”

```java
create mybutton of class JButton using "Push to Quit"
```

The next two steps declare a listener object and create this object, which will respond when the user clicks the button. The actual behavior of the listener object is defined in a class that has to be defined by the programmer. The following statements declare and create the listener object of class `Bquithandler`.

```java
define bhandler of class Bquithandler
... create bhandler of class Bquithandler
```

After the button object and its corresponding listener object are created, the listener object has to be registered as the listener to the button object. Recall that a button generates an action event when pressed, which is handled by action listeners. The following statement invokes function `addActionListener` of the button to register its listener object, `bhandler`. Function `addActionListener` is defined in class `JButton`.

```java
call addActionListener of mybutton using butthandler
```

The button can now be added to the content pane of the window. The following statement adds the button defined previously to the content pane with the south position. The content pane uses the border layout manager in this example.

```java
call add of cpane using mybutton, BorderLayout.SOUTH
```
The following class, `Fpbutton`, completely implements a window with three graphical components: a text label, a label with an icon, and a button. An object listener is defined for the button. The listener object terminates the program when the user clicks the button.

```java
use all javax.swing // Library for graphics
use all java.awt
//
// description
This class creates and displays a frame window with a text label, an image, and a button. */
class Fpbutton is
public
description
This is the main function of the application. */
function main is
constants
  define WIDTH = 400 of type integer
  define HEIGHT = 300 of type integer
objects
  define frame_obj of class JFrame // window
      // content pane
  define cpane of class Container
  define blabel1 of class JLabel // label
  define oosimlabel of class JLabel // for image
  define oosimlimage of class ImageIcon // image
  define quitb of class JButton
      // layout manager
  define lmanager of class BorderLayout
  define bhandler of class Bquithandler
begin
  create frame_obj of class JFrame using "OOSimL"
  create blabel1 of class JLabel using "The OO Simulation Language"
  create quitb of class JButton using "Quit"
  create oosimlimage of class ImageIcon
      using "oosiml.gif"
  create oosimlabel of class JLabel
      using psimimage
  define lmanager of class BorderLayout
  define bhandler of class Bquithandler
  call getContentPane of frame_obj return to cpane
  call setLayout of cpane using lmanager
  // register the listener object with the
```
// button object
    call addActionListener of quitb using butthandler
    // add the text image label and text label
    // components to the content pane of
    // the window
    call add of cpane using oosimlabel,
           BorderLayout.CENTER
    call add of cpane using blabel1,
           BorderLayout.NORTH
    call add of cpane using quitb,
           BorderLayout.SOUTH
    call setSize of frame_obj using WIDTH, HEIGHT
    call setVisible of frame_obj using true
endfun main
endclass Fpbutton

For every type of event object, there is a type of listener object. Therefore, the class definition that implements the behavior of listener objects depends on the type of events that these objects can handle (or respond to). For action events, the class definition must implement interface ActionListener, which is included in package AWT. The only method specified in the interface is actionPerformed, and it has to be completely implemented in the class defined for action listener objects.

The following class, Bquithandler, defines the behavior of the object listener for the button used in class Fpbutton, defined previously.

use all javax.swing     // Library for graphics
use all java.awt.event
description
    This class defines the behavior of listener objects for the button. When the user clicks the button, the program terminates. */
class Bquithandler implements ActionListener is
    public
description
    The only function in the class. */
    function actionPerformed parameters
        define myev of class ActionEvent is
        begin
            call System.exit using 0
        endfun actionPerformed
endclass Bquithandler

A window appears on the screen when the program, which consists of the two
classes defined previously, executes. When the user clicks the button, the program terminates.

17.11 Data Input

A text field component is used for entering data to an application. This component can also display data. When used for input, this component can generate an action event and send it to an action listener. This implies that an action listener object must be created and registered to the text field object.

When a user enters data in a text field and presses the Enter key, the text field object generates an action event and sends it to an action listener object. The text field is an object of class \texttt{JTextField}, which can be created with a given size and default text.

The following two statements declare an object reference and create the corresponding object of class \texttt{JTextField} of size 20 characters.

\begin{verbatim}
define text1 of class JTextField
create text1 of class JTextField using 20
\end{verbatim}

To register a listener object with a text field is similar to registering a listener object to a button. The following statements declare and create a listener object, register an action event listener object with the text field object \texttt{text1}, and include the text field in the content pane.

\begin{verbatim}
define tfieldlistener of class Tlistener
define tfieldlistener of class Tlistener
create tfieldlistener of class Tlistener
call addActionListener of text1 using tfieldlistener
call add of cpane using text1
\end{verbatim}

All data is entered and displayed as strings, so proper conversion needs to be carried out for entering and displaying numerical data. To get string data entered by a user into a text field, method \texttt{getText} of the text field object is used. To display string data in a text field, the method \texttt{setText} is used with the string as the argument. The following statement gets the string data entered by the user in the text field \texttt{text1}, and assigns the string value to variable \texttt{ss}.

\begin{verbatim}
define ss of type string
call getText of text1 return to ss
\end{verbatim}
In a similar manner, the following statement displays string yy on the text field object text1.

define yy of type string
... call setText of text1 using yy

To convert the string value entered in a text field to numeric of type double, method Double.parseDouble is used with the string value as the argument. This is actually a static method parseDouble in class Double, which is a Java library class. The following statement converts the string ss to a numeric (of type double) variable dd.

define dd of type double
... call Double.parseDouble using ss return value to dd

To display numeric data (of type double) to a text field, it must first be converted to a string value. Method String.valueOf (static method valueOf of class String) must be invoked with the numeric value as the argument. The following statement converts variable dd of type double to a string and assigns it to string variable ss.

call String.valueOf using dd return to ss

17.12  Decimal Formatting

For non-integer numeric output, formatting the value to be shown is important; otherwise, too many decimal digits will appear and be shown to the user. The outcome of using a decimal formatter is a string that represents the number in one of several formats. The following statements declare an object variable of class DecimalFormat (a Java library class), create the formatter object using a formatting pattern, and invoke function format of the formatter object using the numeric value to format.

define myformat of class DecimalFormat
... // object for formatting output numeric data
create myformat of class DecimalFormat
   using "###,###.##"
... call format of myformat

© 2014  J. M. Garrido
using dincrease return to form_increase
call format of myformat
using dsalary return to form_salary

In this example, the two numeric variables of type double: dincrease and dsalary, are formatted with the same pattern, which allows only two decimal digits (after the decimal point). The resulting numeric value is normally rounded to the specified decimal digits.

Formatting includes an implied conversion of the noninteger numeric value to a string. This string data can be shown in a text field in the relevant container.

17.13 Applets

Applets are graphical applications that are not standalone programs, because they require a Web browser to run. The code of the compiled class for an applet is placed in an HTML file with the appropriate tags. When a user uses his Web browser to start an applet, the compiled classes of the applet in the HTML file are downloaded from the server and execute.

Suppose the class for an applet is named Aoosimlogo, the appropriate tags in the HTML file with the compiled class are as follows:

```html
<applet
code = "Aoosimlogo.class" width=300 height=400>
</applet>
```

An applet normally includes graphical components in addition to any computation that may appear in a program. A Web browser displays the complete Web page, including the GUI for the applet. A small and complete HTML file with an applet embedded in it is shown next.

```html
<HTML>
<HEAD>
  <TITLE> The OOSIML Applet </TITLE>
</HEAD>
<BODY BGCOLOR=blue TEXT=white>
  This is a simple applet showing an image.
  Any text included here in the HTML document.
  <CENTER>
  <H1> The OOSIML Applet </H1>
  <P>
  <APPLET CODE="Aoosiml.class"
          WIDTH=250 HEIGHT=150>
  </APPLET>
</BODY>
</HTML>
```

© 2014 J. M. Garrido
There are various aspects of an applet to consider when defining the corresponding class, and that differentiates it from a conventional class. In an applet, the class definition must inherit class `JApplet` from the Swing library, or the `Applet` class from the AWT library. As mentioned before, applets are not standalone programs, so function `main` is not used. Instead, function `init` is included. A frame for a window is not defined because the applet automatically constructs a window. The size of the applet window is set in the HTML file. The Web browser makes the applet visible.

Class `Aoosiml` defines an applet that displays an image. It defines three graphical components that are labels. The code with the implementation of the applet class `Aoosiml` is presented as follows.

```java
use all javax.swing // Library for graphics
use all java.awt
description
    This applet creates and displays a frame window
    with an image and a text label. */
class Aoosiml inherits JApplet is
    public
description
    This is the main function of the application. */
    function init is
        objects
            // content pane
            define cpane of class Container
            define blabel1 of class JLabel // text label
            define blabel2 of class JLabel
            define kjplabel of class JLabel // for image
            define kjpimage of class ImageIcon // image
            // layout manager
            define lmanager of class BorderLayout
        begin
            create blabel1 of class JLabel
                using "OOSIML"
            create blabel2 of class JLabel
                using "The Language for OOP"
            create kjpimage of class ImageIcon
                using "kjplogo.gif"
            create kjplabel of class JLabel
                using kjpimage
```
create lmanager of class BorderLayout
call getContentPane return to cpane
call setLayout of cpane using lmanager
// add the text image label and text label
// components to the content pane
call add of cpane using
   kjplabel, BorderLayout.CENTER
call add of cpane using
   blabel1, BorderLayout.NORTH
call add of cpane using
   blabel2, BorderLayout.SOUTH
endfun init
endclass Aossimlogo

To execute the applet, a Web browser is used to run the applet class. To test the applet the appletviewer utility can be used.

17.14 Panel Containers
The windows discussed in previous sections placed components in the content pane of the frame. The content pane is the container where any types of components or small containers are placed.

The largest container defined is an object of class JFrame. The only way to add components to this object is by using its content pane. Smaller containers are objects of class JPanel. These objects can contain components such as labels, buttons, text fields, and other components. With panels (objects of class JPanel), it is possible to organize a GUI in a hierarchical manner. A GUI with several panels is structured in such a manner that all the panels and other components are placed in the content pane of the frame.

The following statements declare two panel objects, create the panels, and set the layout manager for each panel.

define fpanel of class JPanel
define bpanel of class JPanel
   . . .
create bpanel of class JPanel
create fpanel of class JPanel
   . . .
call setLayout of fpanel using gridmanager
call setLayout of bpanel using flowmanager

The various components can be added to each panel, and the panels can be added to the content pane of the frame.

© 2014 J. M. Garrido
For example, consider an application that has two panels that are placed in the content pane of the frame using border layout. The first panel contains the labels and text fields using the grid layout. The second panel contains the two buttons using flow layout. The following class implements the two panes discussed.

```java
use all javax.swing // Library for graphics
use all java.awt
//
description
This program computes the salary increase for an employee. If his/her salary is greater than \$45,000, the salary increase is 4.5%; otherwise, the salary increase is 5%.
Two panels are used in this class. The first is used to place the labels and text fields, the second is used to place the two buttons.
The program uses the following components: buttons, labels, and text fields.
*/
class Psalarygui is
public
description
   This is the main function of the application. */
function main is
   constants
      define WIDTH = 400 of type integer
      define HEIGHT = 300 of type integer
   object references
      define sal_frame of class JFrame
      define cpane of class Container
      define fpanel of class JPanel
      define bpanel of class JPanel
      define namelabel of class JLabel
      define salarylabel of class JLabel
      define agelabel of class JLabel
      define increaselabel of class JLabel
      define nametfield of class JTextField
      define agetfield of class JTextField
      define inctfield of class JTextField
      define salarytfield of class JTextField
      define bordermanager of class BorderLayout
      define gridmanager of class GridLayout
      define flowmanager of class FlowLayout
      define incbutt of class JButton
```
define quitbutt of class JButton
define actlistener of class Sal_listener
begin
create sal_frame of class JFrame using "Salary Problem"
set cp = call getContentPane of sal_frame
create bordermanager of class BorderLayout
create gridmanager of class GridLayout
using 4, 2
create flowmanager of class FlowLayout
create bpanel of class JPanel
create fpanel of class JPanel
create namelabel of class JLabel
using "Enter name: "
call setBackground of fpanel using Color.lightGray
create agelabel of class JLabel
using "Enter age: "
call setBackground of bpanel using Color.blue
create salarylabel of class JLabel
using "Enter salary: "
call setBackground of cpane using Color.orange
create increaselabel of class JLabel
using "Salary increase: "
call setBackground of fpanel using Color.lightGray
create nametfield of class JTextField using 20
call setBackground of bpanel using Color.blue
create agetfield of class JTextField using 20
call setBackground of cpane using Color.orange
create salarytfield of class JTextField using 20
create inctfield of class JTextField using 20
create incbut of class JButton using "Increase"
call setBackground of bpanel using Color.blue
call setBackground of cpane using Color.orange
call add of fpanel using namelabel
call add of fpanel using nametfield
call add of fpanel using agelabel
call add of fpanel using agetfield
call add of fpanel using salarylabel
call add of fpanel using salarytfield
call add of fpanel using increaselabel
call add of fpanel using inctfield
call add of bpanel using incbut
call add of bpanel using quitbutt
create actlistener of class Sal_listener
using salarytfield, inctfield
call addActionListener of incbutt
using actlistener
call addActionListener of quitbutt
using actlistener
call add of cpane using
fpanel, BorderLayout.CENTER
call add of cpane using
bpanel, BorderLayout.SOUTH
call setSize of sal_frame using WIDTH, HEIGHT
call setVisible of sal_frame using true
defun main
defclass Psalarygui

Figure 16 shows the window with two generic panels.

Figure 16: A frame with two panels.

17.15 Drawing Simple Objects

This section presents some simple tools and techniques for drawing graphics objects. The coordinate system used in drawing objects places the origin of the drawing area in its upper-left corner of the drawing area (on the screen). All measures involved in drawing use pixels as the basic unit. The position of a visible dot is measured in the number of pixels to the right of the origin and the number of pixels below the origin. This gives the position using the coordinates (x, y).
17.16 General Functions for Drawing

The general technique for drawing graphics objects is to define a class that inherits class `JComponent` and redefine function `paint`. An object is created, and its reference can then be added to a frame.

Function `paint` is invoked automatically, so there is no need to explicitly call this function. Function `paint` is defined with one parameter, an object reference of class `Graphics`, which is a class in the AWT package. The functions for drawing lines, circles, polygons, and so on, are features of the parameter of class `Graphics`.

To draw a line on the drawing area, from point P1(20, 45) to point P2(75, 50), function `drawLine` is invoked with the coordinates of the two points as arguments. The following statement draws a line from point `P1` to point `P2`, using the object reference, `graph_obj`, of class `Graphics`.

\[
\text{call drawLine of graph_obj using 20, 45, 75, 50}
\]

This and other drawing statements must appear in function `paint`. The following statement draws a rectangle whose upper-left corner is located at point (80, 75), and with 50 pixels for width and 100 pixels for height.

\[
\text{call drawRect of graph_obj using 80, 75, 50, 100}
\]

The following statements draw an arc. The arc is part of an oval that is specified by an enclosing rectangle with the upper-left corner located in \((x, y)\), and the size given by `width` and `height`. The portion of the arc to draw is specified by the starting angle and the final angle (in degrees).

\[
\begin{align*}
\text{define } x &= 20 \text{ of type integer} \\
\text{define } y &= 50 \text{ of type integer} \\
width &= 35 \\
height &= 25 \\
startang &= 0 \\
finalang &= 45 \\
... \\
\text{call drawArc of graph_obj using } \\
x, y, width, height, startang, finalang
\end{align*}
\]

Other drawing functions defined in class `Graphics` are listed in Table 2.
Table 2: Common drawing functions in class Graphics

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawOval</td>
<td>Draws the outline of an oval</td>
</tr>
<tr>
<td>draw2DRect</td>
<td>Draws a highlighted outline of a rectangle</td>
</tr>
<tr>
<td>drawPolygon</td>
<td>Draws a closed polygon</td>
</tr>
<tr>
<td>drawRoundRect</td>
<td>Draws a round-cornered rectangle</td>
</tr>
<tr>
<td>fillArc</td>
<td>Fills a portion of an oval with color</td>
</tr>
<tr>
<td>fillOval</td>
<td>Fills an oval with color</td>
</tr>
<tr>
<td>fillPolygon</td>
<td>Fills a polygon with color</td>
</tr>
<tr>
<td>fillRect</td>
<td>Fills a rectangle with color</td>
</tr>
<tr>
<td>fillRoundRect</td>
<td>Fills a round-cornered rectangle with color</td>
</tr>
<tr>
<td>fill3DRect</td>
<td>Fills a rectangle with color</td>
</tr>
</tbody>
</table>

18 Exceptions and I/O

Robust programs should take specified actions when errors occur. The most common type of runtime errors are division by zero, array indexing out of bounds, variable value out of range, illegal reference, and I/O errors.

An exception is an error or an unexpected condition that occurs while the program executes. The mechanism for detecting these errors or conditions and taking some action is called exception handling.

All input/output is carried out with streams, except with GUIs. A stream is a sequence of bytes; the direction of the flow of data determines whether it is an input or an output stream—either incoming from a source or directed toward a destination.

Input and output files are associated with I/O streams. The various class packages allow the programmer to open, process, and close files.

This section first presents the basic concepts associated with exceptions; second, it applies exception handling in discussing I/O streams and files.

18.1 Dealing with Exceptions

The general approach used in programming for dealing with exceptions is to divide the code into two sections:

1. The first section of code detects an exception. This involves identifying some instruction sequence that might generate or throw an exception.

2. The second section of code takes some action to deal with the exception. This is called handling the exception.

There is a wide variety of exceptions, depending on the abnormal condition that occurs. By default, languages do not handle all types of exceptions. When an exception occurs, the program aborts, and the runtime system displays the type of condition detected. For example: “Exception in thread main ArithmeticException: division by zero.” The runtime system also prints a trace of the function calls.
18.2 Checked and Unchecked Exceptions

There are two basic categories of exceptions: checked and unchecked. The first type of exceptions, checked, can be controlled syntactically. The compiler checks that the exception is handled in the program. These exceptions are very likely to occur in the program.

Some of the methods throw various types of exceptions. These are considered checked exceptions. For example, the following function can throw an exception when invoked.

function mread throws IOException is
    . . .
endfun mread

Unchecked exceptions are very difficult to detect at compile time. These exceptions do not have to be handled in the program.

18.3 Basic Handling of Exceptions

There are two blocks of statements that are needed for detection and processing of exceptions. The first block is called a try block and it contains statements that might generate an exception. When an exception occurs on the try block, the second block, called the catch block, begins to execute immediately.

The parameter in the catch block is an object reference declaration of type Exception. When an exception occurs, this parameter is passed and can be used in the block to get information about the exception. Method getMessage, defined in class Exception, can be used to get a description of the exception. The general syntactic structure of these two blocks of statements is:

\[
\text{try} \begin{align*}
&\begin{align*}
&\text{begin} \\
&\quad \langle \text{statements} \rangle \\
&\text{endtry} \\
&\text{catch} \langle \text{parameters} \rangle \\
&\text{begin} \\
&\quad \langle \text{statements} \rangle \\
&\text{endcatch}
\end{align*}
\end{align*}
\]

The catch block provides a name to the object reference of the exception object that is caught. With this object reference, the message of the exception object can be displayed and/or any other action can be implemented to handle the exception.

A variation of the problem that reads data for employees follows. The solution presented includes an exception that occurs when the user types the value of the age that is zero or negative. Class TestException implements the solution to the problem. The code for this class follows.

© 2014 J. M. Garrido
This program checks for an exception in the value of age. */
class TestException is
public
description
  This is the main function of the application.
  If the age is zero or negative, an exception is thrown and caught. */
function main is
variables
  define obj_age of type integer
  define increase of type float
  define obj_salary of type float
  define obj_name of type string
  define lmessage of type string // message for exception
object references
  define emp_obj of class Employee
  define lexecep_obj of class Exception
begin
  displayn "\n Enter name: "
  read obj_name
  displayn "\n Enter age: "
  read obj_age
  // Check for exception
  try begin
    if obj_age <= 0 then
      create lexecep_obj of class Exception using
        "Exception: age negative or zero"
      throw lexecep_obj
    endif
  entry
  catch parameters define excobj of class Exception
    begin
      call getMessage of excobj return to lmessage
      display lmessage
      display "Retrying . . . "
      displayn "\n Enter age: "
      read obj_age
    endcatch
    // continue with processing
    displayn "\n Enter salary: "

read obj_salary
create emp_obj of class Employee using
    obj_salary, obj_age, obj_name
call sal_increase of emp_obj return to increase
call get_salary of emp_obj return to obj_salary
display "Employee name: ", obj_name
    " new salary: ", obj_salary
endfun main
endclass TestException

In the program discussed, handling of the exception is carried by displaying information about the exception object, lexcep_obj, by invoking its method getMessage, and by executing the instructions that reread the value of age. All these statements appear in the catch block. If no exception occurs, the catch block does not execute, and the program proceeds normally.

When the user types a zero or negative value for the age, an exception is thrown. The exception is detected in the try block. To handle the exception, the statements in the catch block are executed. These statements display the message that explains and identifies the exception and allows the user to reenter the value for the age.

When the program executes and the user enters a “bad” value for the age, the program stops, displays the message, and resumes to let the user reenter the value for the age.

18.4 Files

A disk file organizes data on a massive storage device such as a disk device. The main advantage of a disk file is that it provides permanent data storage; a second advantage is that it can support a large amount of data. A third advantage of a disk file is that the data can be interchangeable among several computers, depending on the type of storage device used. On the same computer, disk files can be used by one or more different programs.

A disk file can be set up as the source of a data stream or as the destination of the data stream. In other words, the file can be associated with an input stream or with an output stream.

18.5 Text and Binary Files

Text files are human-readable. They contain data that is coded as printable strings. A byte is coded as a single text character. For example, an OOSimL source program is stored in a text file with a .osl extension. In the same manner, a Java source program is also stored in a text file. In simple terms, a text file consists of a sequence
of characters. Lines are separated by two characters, carriage return (CR) and line feed (LF); these are placed at the end of a line by pressing the Enter key\(^1\).

A binary file is not human-readable. Its data is stored in the same way as represented in memory. Especially for numeric data, the representation in memory is just ones and zeroes. A compiled program is stored in a binary file.

Binary files take less storage space and are more efficient to process. When reading or writing numeric data, there is no conversion from or to string format.

### 18.6 Handling Text Files

The normal procedure for a program that processes data on a file is the following:

1. Open the file for input or output. This step is called opening the file for input or output; it attaches the file to a stream.
2. Read data from the file or write data to the file.
3. Close the file.

For these file processing tasks, Java provides several library classes. Most of the classes for stream I/O are located in package `java.io`.

### 18.7 Output Text Files

Java provides two predefined Java classes that are used to create objects for opening a text file for output, `FileOutputStream` and `PrintWriter`. The following statements declare the two object references and create the corresponding objects to an output text file called `mydata.txt`.

```java
define myoutfile of class FileOutputStream
define myoutstream of class PrintWriter
.
try begin
  create myoutfile of class FileOutputStream
    using "mydata.txt"
  create myoutstream of class PrintWriter
    using myoutfile
endtry
```

The previous statements have connected the disk file to an output stream, `myoutstream`. This opening of the file could generate an exception if the file cannot be created. For this reason, the statements must appear in a `try` block. To handle the exception, a `catch` block must immediately follow.

\(^1\)In Unix, only LF is placed at the end of the line.
catch parameters object e of class
       FileNotFoundException
       begin
       display "Error creating file mydata.txt"
       terminate
       endcatch

If the output file cannot be created, an exception is raised (thrown) and statements in the catch block display an error message related to the exception and terminate the program.

The output stream created is used for all output statements in the program with methods display and displayn of object reference myoutstream.

After the output file has been created, it can be used to write string data. The numeric data must first be converted to string. For example, to convert an integer value to a string value, function valueOf of the class String is invoked. The following statements declare a variable of type integer, int_val, declare a variable of type string, str_val, convert the value of the integer variable to a string, and assign the value to the string variable, str_val.

variables
       define int_val of type integer
       define str_val of type string

... set str_val = String.valueOf(int_val)

18.8 Input Text Files

Reading data from an input text file implies reading text lines from the input line. If the data items are numeric, then the source string must be converted to a numeric type. Another aspect of input text files is that the program that reads from the file has no information on the number of lines in the text file.

Java provides two predefined Java classes in the io package that are used to create objects for opening a text file for input, BufferedReader and FileReader. The following statements declare the two object references and create the corresponding objects for an input text file called “mydata.txt.”

define myinfile of class BufferedReader
define myreader of class FileReader

... try begin
       create myreader of class FileReader
       using "mydata.txt"
create myinfile of class BufferedReader
    using myreader
endtry

Opening a text file for input can throw an exception if the file cannot be found. Therefore, the statements that create the two objects must appear in a try block. To handle the exception, a catch block must immediately follow.

catch parameters object excep of
    class FileNotFoundException
begin
    display "Error opening file: ", file_name
    terminate
endcatch

In a similar manner to dealing with the output file creation, if the input file cannot be opened, the statements shown in the catch block display an error message and terminate the program.

After opening the file for input, the program can read input streams from the file, line by line. The Java method readLine is invoked to read a line of text data. This method is defined in class BufferedReader. The statement to read a line of data must be placed in a try block because a reading error might occur. The exception is thrown by method readLine.

One additional complication, compared to writing a text file, is that it is necessary to check whether the file still has data; otherwise, there would be an attempt to read data even if there is no more data in the file.

When there is no more data in the file, the value of the text read is null. The following statements define a try block with a while loop, which repeatedly reads lines of text from the text file.

try begin
    // Read text line from input file
    call readLine of myinfile return to indata
    while indata not equal null do
        // get name
        set obj_name = indata
        . . .
        // read next line
        set indata = call readLine of myinfile
    endwhile
endtry
The first statement in the try block reads a text line from the file. All subsequent reading is placed in the while loop.

Because the text file can only store string values, conversion is needed for numeric variables. For example, obj_salary and increase are variables of type double, so the string value read from the text file has to be converted to type double. The following statements read the text line (a string) from the file, and then convert the string value to a value of type double with method Double.parseDouble.

```java
// get salary
call readLine of myinfile return to indata
call Double.parseDouble using indata return to obj_salary

// get increase
call readLine of myinfile return to indata
call Double.parseDouble using indata return to increase
```

Method Integer.parseInt would be invoked if conversion were needed for an integer variable.

### 18.9 Files with Several Values on a Text Line

A text file usually contains lines with several values per line, each separated by white spaces or blanks. In this case, the program must read a line of text from the file, separate and retrieve the individual string values, and convert the values to the appropriate types (if needed). The program must repeat this for every line that it reads from the text file. The following text file has three lines, each with various values, some string values and some numeric values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>James_Bond</td>
<td>54</td>
<td>2331.034</td>
<td>54131.785</td>
</tr>
<tr>
<td>Jose_M._Garrido</td>
<td>48</td>
<td>2104.5376</td>
<td>48872.04</td>
</tr>
<tr>
<td>John_B._Hunt</td>
<td>38</td>
<td>1980.2825</td>
<td>41585.93</td>
</tr>
</tbody>
</table>

The Java class StringTokenizer facilitates separating the individual strings from a text line. The following statements declare and create an object of class StringTokenizer, read a line from the text file, and get two string variables, var1 and var2, from the line.

```java
// declare object ref for tokenizer

object tokenizer of class StringTokenizer
```

© 2014 J. M. Garrido
// read line from text file
set line = call readLine of input_file

// create tokenizer object
create tokenizer of class StringTokenizer using line

// get a string variable from line
set var1 = call nextToken of tokenizer

// get another string variable from line
set var2 = call nextToken of tokenizer

To get the number of substrings remaining on a line, the Java method countTokens can be invoked with the tokenizer object. This function returns a value of type integer. A similar function, hasMoreTokens, returns true if there are substrings on the line; otherwise, it returns false.

Class Lfileproc is similar to class Rfileproc, but it reads a line from the text file and separates the individual string variables for name, age, increase, and salary.